From safety analysis to experimental validation by fault injection - Case of automotive embedded systems

Ludovic Pintard

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Informatique et systèmes embarqués

Présentée et soutenue par :

M. LUDOVIC PINTARD

le jeudi 28 mai 2015

Titre :

DES ANALYSES DE SECURITE A LA VALIDATION EXPERIMENTALE PAR INJECTION DE FAUTES - LE CAS DES SYSTEMES EMBARQUES AUTOMOBILES

Ecole doctorale :

Systèmes (Systèmes)

Unité de recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (L.A.A.S.)

Directeur(s) de Thèse :

MME KARAMA KANOUN

M. JEAN CHARLES FABRE

Rapporteurs :

M. LAURENT PAUTET, TELECOM PARISTECH

M. PÉDRO-JOAQUIN GIL VICENTE, UNIVERSITAT POLITECNICA DE VALENCE

Membre(s) du jury :

M. GILLES MOTET, INSA TOULOUSE, Président

M. JEAN CHARLES FABRE, INP TOULOUSE, Membre

M. MICHEL LEEMAN, SOCIETE VALEO, Membre

M. PHILIPPE QUERE, RENAULT GUYANCOURT, Membre

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
Alan Turing, 1950

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés dans le cadre d'une thèse CIFRE entre le *Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS)* du *Centre National de la Recherche Scientifique (CNRS)* de Toulouse, et du groupe *Valeo*.

Je remercie M. Jean Arlat qui assure la direction du *LAAS-CNRS* de m'avoir accueilli au sein de ce laboratoire. Je tiens aussi à remercier Mme Karama Kanoun et M. Mohammed Kaaniche, responsables successifs de l'équipe *Tolérance aux fautes et Sûreté de Fonctionnement (TSF)* du *LAAS-CNRS* dans laquelle j'ai réalisé mes travaux de thèse.

Je tiens à remercier M. Xavier Levesque, directeur du *Group Electronics Expertise and Development Services (GEEDS)*, ainsi que M. Paul Degoul, responsable *GEEDS Software/System/Safety/Simulation* de m'avoir accueilli dans leur service au sein du groupe *Valeo*.

J'exprime ma profonde gratitude à M. Gilles Motet, Professeur à l'*INSA Toulouse*, pour l'honneur qu'il me fait en présidant mon jury de thèse, ainsi qu'à :

- M. Pedro Joaquín Gil-Vicente, Professeur, *Universidad Politechnica de Valencia*,
- M. Laurent Pautet, Professeur, *TELECOM Paris*,
- M. Philippe Quéré, Team Leader, *Renault*,
- M. Michel Leeman, Master Expert, *Valeo*,
- M. Jean-Charles Fabre, Professeur *INP Toulouse*,
- Mme Karama Kanoun, Directeur de Recherche, *LAAS-CNRS*,
- Et M. Matthieu Roy, Chargé de recherche, *LAAS-CNRS*,

d'avoir accepté de participer à mon jury de thèse.

Je remercie tout particulièrement MM. Pedro Joaquín Gil-Vicente et Laurent Pautet qui ont accepté la charge de rapporteur.

J'exprime ma très sincère reconnaissance à Mme Karama Kanoun, Directeur de Recherche *CNRS*, et MM. Jean-Charles Fabre et Matthieu Roy, respectivement Professeur de l'*Institut National Polytechnique de Toulouse (INPT)* et Maître de Conférences, pour m'avoir encadré, soutenu, et encouragé tout au long de cette thèse. Je les remercie pour leurs conseils, leur soutien et leur disponibilité. Leurs expériences et leur expertise ont été essentielles au bon déroulement de mes travaux.

Concernant mon encadrement industriel, je tiens à exprimer toute ma gratitude à M. Stéphane l'Hostis, pour m'avoir accueilli dans le *Département Safety* du *GEEDS* à Créteil.

Je tiens à remercier tout particulièrement M. Michel Leeman pour son investissement durant ma thèse, son soutien continu, sa disponibilité et ses conseils avisés. Je tiens à lui exprimer toute ma gratitude pour le temps qu'il m'a accordé et sa pédagogie qui m'ont été très utiles. Je tiens aussi à remercier M. Abdelillah Ymlahi-Ouazzani, je lui suis reconnaissant du temps et de sa patience pour tous les problèmes de mise en œuvre qu'il a contribué à résoudre, ainsi que de la pertinence de ses remarques. Merci également à Youness Kamel pour son aide à l'implémentation de l'outil de test.

Je tiens aussi à remercier toute l'équipe : Nieves, Abraham, Elmahdi, Abdelillah, Gilles, Ryad, Nabila, Annabelle, Nicolas, Florent, Sylvain, Joris, Mohamed et Styven pour l'accueil qu'ils m'ont réservé, leurs conseils, et pour les moments que nous avons partagés.

J'adresse un merci très spécial à Ivan Studnia, Pierre André, Hélène Martorell, Camille Fayollas et Yann Bachy avec qui j'ai passé de très bons moments durant ces trois années et qui ont eu la gentillesse de pardonner mes plus mauvaises blagues. Merci à Abraham Cherfi pour son soutien et particulièrement sa précieuse aide durant la phase de rédaction du manuscrit. Enfin, je souhaite adresser mes remerciements à Kalou Cabrera Castillos pour son aide dans la préparation de la soutenance.

Merci enfin à l'ensemble des doctorants *TSF* avec lesquels j'ai partagé de très bons moments, Ivan, Hélène, Mathilde, Thibault, Pierre, Yann, Joris, Camille, Benoit, Roberto, Carla, Quynh Anh, Miguel, Moussa, Kossi, Maxime, Miruna, Olivier, Anthony, Amira et celles/ceux que j'ai peut-être oublié de nommer.

Enfin, je remercie ma famille et mes amis qui sont essentiels dans la réussite de mes projets et l'accomplissement de ce travail. Je remercie plus particulièrement Nicolas et Diane. *« Last but not least »,* je remercie mes parents, dont la patience et le soutien ont été sans faille.

Abstract

Due to the rising complexity of automotive Electric/Electronic embedded systems, Functional Safety becomes a main issue in the automotive industry. This issue has been formalized by the introduction of the ISO 26262 standard for functional safety in 2011. The challenges are, on the one hand to design safe systems based on a systematic verification and validation approach, and on the other hand, the fulfilment of the requirements of the ISO 26262 standard. Following ISO 26262 recommendations, our approach, based on fault injection, aims at verifying fault tolerance mechanisms and non-functional requirements at all steps of the development cycle, from early design phases down to implementation.

Fault injection is a verification technique that has been investigated for a long time. However, the role of fault injection during design phase and its complementarities with the experimental validation of the target have not been explored. In this work, we investigate a fault injection continuum, from system design validation to experiments on implemented targets. The proposed approach considers the safety analyses as a starting point, with the identification of safety mechanisms and safety requirements, and goes down to the validation of the implementation of safety mechanisms through fault injection experiments. The whole approach is based on a key fault injection framework, called FARM (Fault, Activation, Readouts and Measures).

We show that this approach can be integrated in the development process of the automotive embedded systems described in the ISO 26262 standard. Our approach is illustrated on an automotive case study: a Front-Light system.

Keywords: Fault Injection, Automotive, Embedded Systems, Safety, Verification, and ISO 26262

Résumé

En raison de la complexité croissante des systèmes automobiles embarqués, la sûreté de fonctionnement est devenue un enjeu majeur de l'industrie automobile. Cet intérêt croissant s'est traduit par la sortie en 2011 de la norme ISO 26262 sur la sécurité fonctionnelle. Les défis auxquelles sont confrontés les acteurs du domaine sont donc les suivants : d'une part, la conception de systèmes sûrs, et d'autre part, la conformité aux exigences de la norme ISO 26262. Notre approche se base sur l'application systématique de l'injection de fautes pour la vérification et la validation des exigences de sécurité, tout au long du cycle de développement, des phases de conception jusqu'à l'implémentation. L'injection de fautes nous permet en particulier de vérifier que les mécanismes de tolérance aux fautes sont efficaces et que les exigences non-fonctionnelles sont respectées.

L'injection de faute est une technique de vérification très ancienne. Cependant, son rôle lors de la phase de conception et ses complémentarités avec la validation expérimentale, méritent d'être étudiés. Notre approche s'appuie sur l'application du modèle FARM (Fautes, Activations, Relevés et Mesures) tout au long du processus de développement. Les analyses de sûreté sont le point de départ de notre approche, avec l'identification des mécanismes de tolérance aux fautes et des exigences non-fonctionnelles, et se terminent par la validation de ces mécanismes par les expériences classiques d'injection de fautes.

Enfin, nous montrons que notre approche peut être intégrée dans le processus de développement des systèmes embarqués automobiles décrits dans la norme ISO 26262. Les contributions de la thèse sont illustrées sur l'étude de cas d'un système d'éclairage avant d'une automobile.

Mots-clés : Injection de fautes, Automobile, Systèmes Embarqués, Sécurité, Vérification, et ISO 26262

Contents

List of Figures

List of Tables

Glossary

ABS	Anti-lock Braking System
ADAS	Advanced Driver Assistance Systems
API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
AUTOSAR	Automotive Open System Architecture
BSW	Basic Software
CAN	Controller Area Network
CCF	Common Cause Failures
COTS	Component Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAL	Design Assurance Level
DC	Diagnostic Coverage
DFA	Dependent Failure Analysis
DIO	Digital Input Output
DTI	Diagnosis Time Interval
E/E	Electric/Electronic
EASIS	Electronic Architecture System Engineering for Integrated Safety Systems
ECU	Electronic Control Unit
EDC	Error Detection Coverage
EPS	Electronic Power Steering
ERC	Error Recovery Coverage
ESCL	Electronic Steering Column Lock
ESP	Electronic Stability Program
FFI	Freedom From Interferences
FI	Fault Injection
FIA	Fault Injection Analysis
FIE	Fault Injection Experiment
FMEA	Failure Mode Effects Analysis
FMECA	Failure Mode Effects and Criticality Analysis
FMEDA	Fault Mode and Effect Diagnosis Analysis
FMF	Fault Management Framework
FSC	Functional Safety Concept

FTA	Fault Tree Analysis
HiL	Hardware in the Loop
HMI	Human-Machine Interface
HW	Hardware
I/O	Input/Output
LFM	Latent Fault Metric
LIN	Local Interconnect Network
MCU	Micro Controller Unit
MiL	Model in-the-Loop
MMU	Memory Management Unit
MPU	Memory Protection Unit
N/A	Not available or Not applicable
NA	Not Applicable
OEM	Original Equipment Manufacturer
OS	Operating System
OSEK, OSEK/VDX	Offene Système und deren Schnittstellen für die Elektronik im Kraftfahrzeug, "Open Systems and the Corresponding Interfaces for Automotive Electronics"
PHA	Preliminary Hazard Analysis
PiL	Processor in-the-Loop
PMHF	Probabilistic Metric of Hardware Fault
QM	Quality Management
RAM	Random Access Memory
ROM	Read-Only Memory
RT	Reaction Time
RTE	Run Time Environment
RTOS	Real-Time Operating System
SEU	Single Event Upset
SG	Safety Goal
SiL	Software in-the-Loop
SM	Safety Mechanism
SPFM	Single Point Fault Metric
SW	Software
SW-C	Software Component
SWIFI	Software-Implemented Fault Injection
TTI	Tolerance Time Interval
UE	Undesired Event
V&V	Verification and Validation

Introduction

The criticality of automotive embedded systems is becoming a major issue. Indeed, the growing complexity of these systems due to the integration of more functionalities as well as the integration of mixed criticality functionalities into electronic systems, may lead to hazardous behavior if the development process is not improved. The introduction of comfort system (*e.g.*, Electronic Power Steering—EPS), active safety systems (*e.g.*, airbags, brake assist) and, in a near future, of autonomous cars necessitates more stringent verification and validation methods.

Safety is a major issue in the automotive domain, due to the cost of vehicle recall when a critical defect is discovered. Major car manufacturers have been confronted to these massive recalls (Shepardson, 2015) (Strong, 2015). In addition, several class actions sued car manufacturers (BARR Group, 2014) (Koopman P. , 2014) for defect of electric/electronic devices. Hence, activities must tend toward a rigorous development process, which closely integrates safety design activities (*e.g.*, definition of safety requirements, definition of safety mechanisms) and the verification activities.

The introduction of ISO 26262 standard for functional safety, in 2011, in the automotive industry, is an important step in this direction. The ISO 26262 standard proposes methods and techniques that should be integrated in the development process in order to ensure safety. ISO 26262 notably highlights that fault injection should be used in the development process. All the verification and validation activities are impacted, even for the verification of the design. This recommendation raised the issue of the role of fault injection in the design phase, which is, as far as we know, a difficult problem that has not been investigated yet.

Fault Injection is a verification technique that has been investigated for a long time (end of the 80's - early 90's). Today, fault injection has been applied to many different targets: Operating System – OS, Middleware, web services, web servers, embedded systems, *etc.* The results of fault injection campaigns are twofold: the verification of the fault tolerance mechanisms, with the estimation of error detection and error recovery coverage, and the experimental evaluation of the robustness of the target, *i.e.* the identification of the failure modes.

In this dissertation, we investigate a fault injection continuum, from system design validation to experiments on implemented targets. The proposed approach considers the safety analyses as a starting point, with the identification of safety mechanisms and safety requirements, and goes down to the validation of safety mechanisms implementation with fault injection experiments.

Previous work performed on implemented targets has shown the relevance of the FARM fault injection model. **FARM** stands for **F**ault, **A**ctivation, **R**eadouts and **M**easures. FARM is a key concept enabling a precise definition of fault injection experiments on implemented targets. Most fault injection studies are based on this model and all studies are compatible with this model.

Our study starts with the investigations of the two following questions: **Can the FARM method be applied at the early design phase? What are the expected fault injection's outcomes in the early validation of safety requirements?**

As we will see, these two questions can be refined:

- What are **the targets?** Can we use the models as targets?
- What are **the measures?** Is the final aim to check that a safety mechanism exists, or to look for possible violations of a given property?
- What is **the fault model?** Do we define it from system design, which results in an abstract fault model, or from system semantics, *i.e.*, application-oriented?
- What does **activation** mean**?** How behavioral description, defining when fault injection is triggered, can be provided? Should we derive it from use cases, state diagrams or sequence diagrams?

A deep analysis of these questions led us to **develop an approach covering the whole development process**, which enables the validation of critical embedded. Our approach shows the link between safety analysis and the application of fault injection in a seamless fashion; it shows the complementarities of both approaches in the design and validation process. We show that this approach can be part of the development process of the automotive embedded systems described in the ISO 26262 standard.

This dissertation is structured in six chapters.

Chapter 1 and Chapter 2 present definitions and general notions about automotive systems, dependability, and development process.

Chapter 1 discusses dependability notions (dependability attributes, threats and means) and their applicability in the automotive industry, particularly in the context of the ISO 26262 standard. Finally, we focus on a specific verification and validation method: Fault Injection. The state of the art of Fault injection addresses the different objectives of this method, the developed techniques and tools that have been developed and the recent automotive studies related to this topic.

Chapter 2 aims at highlighting the integration of fault injection into the development cycle of an automotive system in the context of ISO 26262. We describe separately the activities of the functional development process and the safety development process. Then, we discuss the impacts and the objectives of fault injection in the various phases of the development cycle. This discussion raises the main issue of the thesis: the continuous application of fault injection activity all along the development cycle of an automotive embedded system.

This question is answered in Chapter 3 and Chapter 4. Our approach enables to manage fault injection in all phases of the development cycle, beginning in the pre-implementation phase (Chapter 3) and ending by the post implementation phase (Chapter 4). The meaning of fault injection during this phase is investigated using FARM model as a framework in all phases. We show the complementarities be-

tween the safety analyses and fault injection. In addition, we show how fault injection experiments should be guided using the results of pre-implementation phases, and we discuss the measures obtained in the post-implementation phase on the analyses of the pre-implementation phase.

Chapter 5 and Chapter 6 illustrate the whole methodology on a case study, from analyses to experiments. The case study is a Front-Light System, which controls the low-beam headlights of the vehicle. Chapter 5 applies the proposed approach of Chapter 3 and Chapter 4 by performing FIA and identifying the fault injection experiments. In Chapter 6, the experiments, defined in Chapter 5, are performed on a prototype using a fault injection tool developed during the thesis.

Finally, we conclude by reminding the main problem addressed, and our principal achievements in dealing with it and recommendations. Possible directions for the future research developments are also presented.

Chapter 1 STATE OF THE ART & CONTEXT

In this chapter, our objective is to describe the overall context of this study. We recall here the recent evolution of automotive embedded system, together with standardization of the development process, particularly for safety. Then, we summarize the basic concepts of dependability. These concepts are linked with safety issues and specific terminology of the automotive systems. Finally, the validation technique, *i.e.*, fault injection, is characterized by presenting its approaches, methods and techniques.

1.1 Electric/Electronic Embedded Systems (E/E Systems)

1.1.1 Automotive E/E Systems

Since the end of the 90's, the automotive industry has changed its way to design vehicles and the underlying systems that compose a vehicle. Back then, the systems were designed following a federal architecture where a single ECU was dedicated to one function or service.

The innovation pace has risen quite rapidly, particularly regarding electronic and computing facilities that lead to replace mechanic and hydraulic commands by electronic components. Before that, each function/system of a car was developed independently from the others.

Today's embedded systems cover a large spectrum of automotive systems: motor control (*e.g.*, fuel injection), passive safety (*e.g.*, airbags), braking systems (*e.g.*, Anti-Lock Blocking System – ABS, Electronic Stability Control - ESP), steering (*e.g.*, Electronic Power Steering - EPS).

These systems exhibit now the following properties:

- systems are **interconnected.** Microcontrollers (or Electronic Control Unit ECU) of the vehicle communicate with each other.
- functions/services are **integrated** in complex systems. A system provides several functions, *e.g.*, the Body Controller of the vehicle controls windows, lights, immobilizes the vehicle, *etc.*
- functions are **distributed** on multiple systems. Several parts of a function are hosted by different systems (microcontrollers). For example, the steering column locking system, or the air conditioning system are distributed.

The main advantage of these solutions is the reduction of the number of ECU in the vehicle. However, it increases significantly the complexity of each ECU. The development efforts are larger and the development process must be improved in order to ensure a correct behavior of the system, particularly regarding dependability aspects.

1.1.2 Standardization Needs: ISO 26262

The integration of E/E systems raised the problem of the coexistence of functions or services having different levels of criticality in a single system. Indeed, current systems integrate both critical and non-critical functions. A critical function can lead to an Undesired Event—UE, *i.e.*, an accident in the worst case. In addition, many actors are involved in the development process of a car: a car manufacturer (Original Equipment Manufacturer – OEM), and several suppliers (Tier 1, Tier 2) which develop products for the system defined by the OEM. Each company has its own development process, therefore it is necessary to define and follow robust design rules in order to justify work methods and documentation at all development steps. Hence, all activities ensuring dependability have to be traced.

It is worth noting that there are neither regulations nor directives on functional safety in the automotive industry. Besides, there is no legal requirement for certification of automotive E/E systems. First, several actors decided to adhere (voluntary) to the state of the art defined in the IEC 61508 (IEC 61508, 2010). Contrary to ARP 4754/ED-79 (SAE International, 2010) **Guidelines For Development Of Civil Aircraft and Systems**, ED-12#/DO-178#(RTCA & EUROCAE, 2011) (with # = A in 1985, B in 1992 and C in 2011) for **Software Considerations in Airborne Systems and Equipment Certification**, or the safety guides (*e.g.* 50-SG-D3 and 50-SG-D8) in nuclear industry, this standard is not reserved to only one domain. Indeed, it proposes an approach applicable to generic embedded systems. The IEC 61508 standard focuses on the overall development process of a system and the steps that have to be respected in order to achieve safety. Particularly, it defines achievable goals for the specification, the design, the implementation and the assessment of Electrical/ Electronic/ Electronic Programmable Systems (E/E/EP).

Since 2011, a derived version called ISO 26262 (ISO 26262, 2011) is used. This standard is the result of a joint work between the major actors of the automotive domain aiming at specifying best practices for the documentation, the interactions between actors and the methods and techniques to justify functional safety of systems. This standard facilitates exchanges between OEMs and suppliers by exhibiting requirements to achieve.

The scope of the ISO 26262 is the **functional safety**, *i.e.*, *"the absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems"*. "*Non functional safety"* aspects are out of the scope of the study, *e.g.*, a cause of a malfunction could not be a fire caused by external conditions, such that a humid environment on an E/E system, or an electrical shock with a contact to a high voltage source. Instead, functional safety covers fire due to an over excitation of an alternator within the system in operation (design bug, aging of wires, *etc.*).

The ISO 26262 is divided in ten parts as described in Figure 1.1.

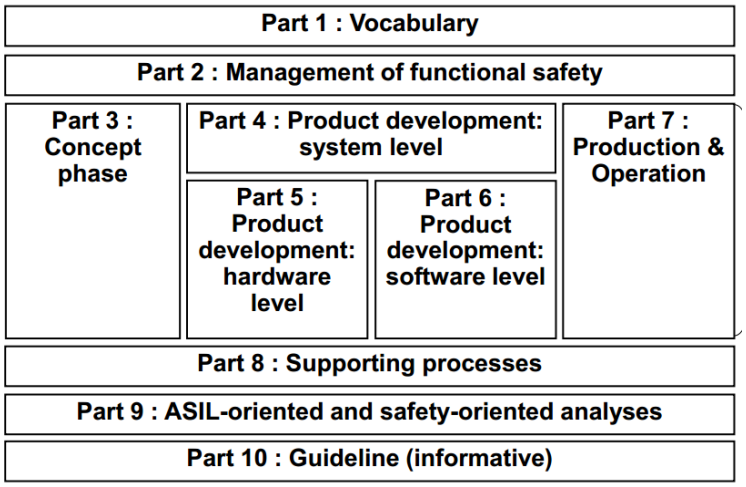


FIGURE 1.1 THE TEN PARTS OF THE ISO 26262 (ISO 26262, 2011)

Our work deals with Parts 4, 5 and 6, which provide all the requirements for the development of an automotive system. However, other parts are also very helpful for understanding these requirements: Parts 1, 8, 9 and 10.

1.2 Basic Concepts of Dependability & ISO 26262

The definition of dependability emerged from the work done in the IFIP WG10.4 working group on *Dependable Computing and Fault Tolerance*: Dependability is defined as the *"trustworthiness of a computing system which allows reliance to be justifiably placed on the service to deliver"*. (IFIP WG 10.4, 2015)

Dependability is a key concept for any critical system. It could be seen as the aptitude to avoid the failures that occur during service delivery. The service corresponds to behavior perceived by the users (human or not) or services in interaction with it.

Dependability is also a well-documented concept, and a complete taxonomy can be found in (Avizienis, Laprie, Randell, & Landwehr, 2004). Indeed, dependability is defined by six attributes, three threats and four categories of means.

1.2.1 From Dependability Attributes to Automotive Safety Integrity Levels

1.2.1.1 Dependability Attributes

Dependability encompasses the following attributes, which characterize the quality of the delivered service:

- **Availability**: readiness for correct service;
- **Reliability**: continuity of correct service;
- **Safety**: absence of catastrophic consequences on the user(s) and the environment;
- **Confidentiality**: absence of unauthorized disclosure of information;
- **Integrity**: absence of improper system alterations;
- **Maintainability**: ability to undergo modifications and repairs.

Depending on the industrial field, the significance of each attribute varies. This choice relies on the objectives that should be achieved for a given service. For example, in transportation fields, reliability and safety are of prime priority; in communication system, availability, reliability and confidentiality are the target attributes.

Historically, in automotive industry, the effort was on the achievement of reliability and availability. The improvement of the reliability of components was sufficient to improve the quality of service. Then, the growing complexity and the criticality of E/E systems lead to focus on safety. Today, security importance is rising quickly, in parallel with car's connectivity. In the following of this work, we will mainly concentrate on ***safety*** aspects.

1.2.1.2 Safety & Automotive Safety-Integrity Level

Considering the dependability attributes, the actor of a given domain can define a scale of criticality for the given attribute. Indeed, all systems should be developed correctly! However, depending on their level of criticality, they do not require the same development efforts, in terms of both design and validation. For example, car audio and video systems do not require the same safety effort than a fuel injection system.

The ISO 26262 standard introduces the concept of Automotive Safety Integrity Level (ASIL). They are four levels: from ASIL A (the less critical) to ASIL D (the most critical). There is also a level,

noted Quality Management (QM), which is not associated with any specific requirements. Hence, no safety-related activities are required by ISO 26262 in this case. The ASIL is determined by the highest criticality of hazards, situations at vehicle level that may lead to harm person the system is interacting with.

When assigning these levels, three parameters must be taken into account, see Table 1.1:

1. **severity** that is based on the seriousness of injuries caused by incidents or accidents (S1: Light and moderate injuries, S2: Severe and life-threatening injuries (survival probable), S3: Life-threatening injuries (survival uncertain), fatal injuries);
2. **probability of exposure.** Occurrence of the use case: E1: very low probability, E2: Low probability E3: Medium probability, E4: High probability;
3. **controllability**. It is a subjective concept that is based on the abilities of the "*road user*" (*e.g.*, drivers, pedestrians, *etc.*) to handle the hazard (C1: Simply controllable, C2: Normally controllable, C3: Difficult to control or uncontrollable).

The objective of these criticality levels is to quantify the level of "trust" at which the system should be designed to provide its functions correctly. The more safety critical the system is, the higher the ASIL is, resulting in stringent efforts to comply with the standard.

TABLE 1.1 DEFINITION OF THE SAFETY-ASIL MATRIX (ISO 26262, 2011)

		Controllability		
Severity of the harm	Probability of exposure	C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	ASIL A
	E4	QM	ASIL A	ASIL B
S2	E1	QM	QM	QM
	E2	QM	QM	ASIL A
	E3	QM	ASIL A	ASIL B
	E4	ASIL A	ASIL B	ASIL C
S3	E1	QM	QM	ASIL A
	E2	QM	ASIL A	ASIL B
	E3	ASIL A	ASIL B	ASIL C
	E4	ASIL B	ASIL C	ASIL D

Due to the imperfections inherent to all systems, dependability attributes have to be interpreted in a relative sense, not in an absolute, deterministic one. The requirements for the attributes are therefore specified according to levels and some of them may not be required for a given system.

1.2.2 From Dependability Threats to Fault Model

1.2.2.1 Dependability Threats

The threats to dependability are faults, errors, failures. A **service failure**, also abbreviated to **failure**, is an event that occurs when the service delivered by the implemented system function deviates from the correct service. Hence, it affects the targeted level of satisfaction of one or several dependability attributes. They are linked within the chain of dependability threats illustrated in Figure 1.2.

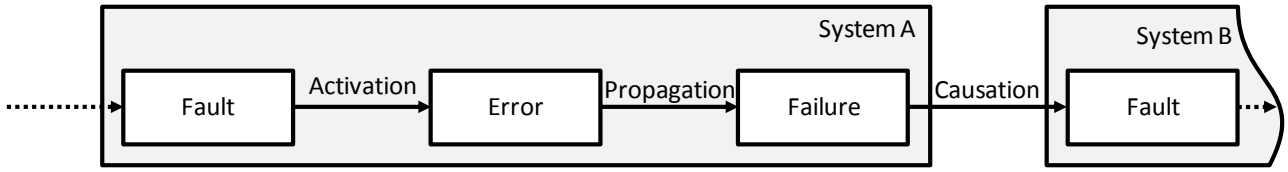


FIGURE 1.2 RECURSIVE CHAIN OF DEPENDABILITY THREATS

At the beginning of a service failure, there is a ***fault,*** *i.e.* the origin of a potential failure. A fault is a defect that can be internal or external to a system. They have been classified into three major overlapping categories (Avizienis, Laprie, Randell, & Landwehr, 2004):

- **development faults**, that include all fault classes occurring during development;
- **physical faults**, that include all fault classes that affect hardware;
- **interaction faults**, that include all external faults.

Although a system may contain a fault, its input and state conditions may never cause this fault to be activated so that an error occurs; in this case, the fault is referred as **dormant**. Then, as soon as a fault has been activated, it produces an **error**, *i.e.*, a part of the system state that may cause a subsequent service failure. The failure occurs when a propagating error –internal **propagation**– reaches and alters the interface of the considered system service.

Finally, the failure may propagate to the interface of another system service (external propagation), that appears as an external fault to this service. It is referred to as **causation**.

1.2.2.2 Fault Model in Automotive Embedded Systems

An automotive embedded system may fail in operation due to either physical faults (hardware aging, EMC, *etc.*) or residual bugs from design or development phase.

Regarding the faults of systems and specific hardware elements, a classification is proposed in the Annex D of ISO 26262-5 (ISO 26262, 2011). In the table, given in APPENDIX 1, each type of element is considered: E/E System, relays, communication links, sensors, processing units, *etc.*

Then, for each component, a set of typical faults, errors or failures of hardware is described. The proposed listing *"does not claim exhaustiveness and can be adjusted based on additional known faults or depending on the application"*. This is intended to provide a representative guideline of the fault model that should be considered in the automotive domain. For example, sensor fault model encompasses faults such as stuck-out of range, stuck-in range, oscillations and offsets.

This table also proposes a guideline for the diagnostic coverage achievable by a safety mechanism. According to this table, a safety mechanism that covers a category (each component has three categories) of faults has the ability to achieve low (60%), medium (90%) or high (99%) diagnostic coverage. For example, a safety mechanism covering all the sensor faults described previously can pretend to achieve a **high**, 99% DC. *"Input comparison/voting (1oo2, 2oo3 or better redundancy"* is a proposed measure to achieve this high DC.

It should be noted that a similar classification of the Appendix D of ISO 26262-5 has been also done in the Electronic Architecture and System engineering for Integrated Safety systems—EASIS European project (Lu, 2009a). However, EASIS classification does not propose an achievable DC level.

Specifically, in software applications, physical faults are modeled as permanent fault (leading to hang or crash) and transient faults (*e.g.* bit-flips and stuck-at in the code and data memory segments leading to value errors). Such faults are always possible due to the aggressive environment of automotive applications and the increasing complexity of the hardware components and system architecture.

To take into account such aggressive environments and complex architectures, ISO 26262-6 (highly) recommends injection of arbitrary values (*e.g.*, by corrupting values of variables, by introducing code mutations, or by corrupting values of CPU registers).

Regarding software faults, also called systematic faults, they may occur due to non-respected rules during the design. The errors could be introduced in system, hardware or software design, because of a misinterpretation of the specifications. In software, the following are potential causes of these design errors: wrong temporal design (sizing, execution order, *etc.*), wrong resource sizing, wrong data usage (wrong choice of data for usage, wrong handling of a data, *etc.*) or non-expected modes. These bugs are introduced during manual coding, or with compiler or linker's default.

1.2.3 From Dependability Means to Verification

Finally, dependability means, whose objective is to ensure dependability attributes from dependability threats, are grouped in four categories:

- **fault prevention** aims to prevent the occurrence or introduction of faults;
- **fault tolerance** aims to avoid service failure in the presence of faults;
- **fault removal** aims to reduce the number and the severity of faults;
- **fault forecasting** aims to provide an estimation of the present number of faults, future incidence and possible consequences of faults.

Fault prevention is ensured by quality control techniques, such as adherence to design rules, throughout the development and the manufacturing process of the system.

1.2.3.1 Fault Tolerance

In order to prevent a service failure, a fault tolerant system needs to integrate in its design error handling techniques, including *error detection*, *error correction*, *error recovery*, *redundancy* and *diversification* (for systematic faults).

All these techniques, when integrated in a design, provide a fault tolerant architecture against predefined faults. Various architectures have been studied and each industrial domain has developed solutions that meet its constraints. For example, triplication with majority vote is a common solution for railways, avionic or aerospace systems since decades, contrary to the automotive domain where this robust solution is usually not necessary, and fail-safe designs are used.

1.2.3.2 Fault Forecasting

Fault forecasting is conducted by performing an evaluation of the system behavior with respect to fault occurrence or activation. The evaluation is composed of two aspects:

- **qualitative**, or ordinal, **evaluation**. This aims at identifying, classifying, ranking the failure modes, or the event combinations (components failures or environmental conditions) which may lead to system failures,

- **quantitative**, or probabilistic, **evaluation**. This aims at evaluating in terms of probabilities the extent to which some of the attributes of dependability are satisfied; those attributes are then viewed as measures of dependability. This evaluation is based on the alternation of correct and incorrect service delivery, to define reliability, availability and maintainability measures.

Particularly in the automotive domain, specific metrics have to be calculated: Probabilistic Metric of Hardware Failure (**PMHF**), Single Point Fault Metric (**SPFM**) and Latent-Fault Metric (**LFM**).

Numerous methods enable to evaluate qualitative and quantitative aspects. Table 1.2 lists safety analysis methods.

TABLE 1.2 SAFETY ANALYSES METHODS

Safety analysis method	Qualitative	Quantitative	Automotive Industry Specific Information
HA&RA[1]	✓		Also referred to as Preliminary Hazard Analysis (PHA) in automotive industry
FME(C)A[2]	✓		✓
FMEDA[3]	✓	✓	Specific to automotive industry. Enables the calculation of architectural metrics from the ISO 26262 (SPFM and LFM)
DFA[4]	✓		Analyzes independence, and non-interference between elements of the component
CPA	✓		
FTA[5]	✓	✓	✓
RBD[6]	✓	✓	
Markov Chain[7]		✓	
Stochastic Petri Nets		✓	
ETA[8]	✓	✓	

Then, the evaluation of the measures can be performed using modeling and analyses but also through testing. This experiment-based approach is tackled with **fault injection** techniques.

1.2.3.3 Fault Removal

Here, we focus on the fault removal activities during development phase. In this phase, the objective is to perform preventive or corrective maintenance, by patching software, replacement of electronic devices, *etc.*

The fault removal activity consists mainly in a verification process, which leads to diagnose the threats and finally to proceed to the necessary corrections. Then, the process must be repeated in order to check that the fault removal process has not inserted new faults. In practice, this step is referred to as **non-regression** verifications.

[1] Hazard Analysis & Risk Assessment (*e.g.*, HAZOP) (M2OS, 2014)
[2] Failure Mode, Effect (and Criticality) Analysis (Bouti & Kadi, 1994), (Department of the Army, 2006), (ECSS-Q-30-02B, 2008)
[3] Failure Mode, Effect and Diagnosis Analysis (L'Hostis, 2013)
[4] Dependant Failure Analysis (ISO 26262, 2011)
[5] Fault Tree Analysis (Barlow & Lambert, 1975)
[6] Reliability Block Diagram (SaRS: Safety and Reliability Society, 2011)
[7] (SaRS: Safety and Reliability Society, 2011)
[8] Event Tree Analysis (M2OS, 2014)

Verification techniques can be classified according to whether or not they involve exercising the system. One the one hand, if the system is not activated these are called **static verification techniques**. Static analyses can be performed manually or automatically. Manual ones are "inspections", "reviews", "walkthrough" techniques (Aurum, Petersson, & Wohlin, 2002), consisting in a detailed analysis of a system artifact (specifications, design, source code, *etc.*). Even if this technique is time consuming, a large number of faults can be identified prior to any execution of the system. Automatic ones based on software tools give informative metrics or lists of anomalies. Static analyses also include theorem proving (requires formal specifications in this case) and model-checking techniques.

On the other hand, the **dynamic verification techniques** of the system are usually referred to as **testing**. A test aims at providing inputs to a system, and verifying that the observed behavior is correct with respect to the specifications. Due to complexity of modern automotive systems, it is not manageable to verify exhaustively a system (except for very specific simple cases). Indeed, a test campaign does not provide a proof of the zero-default behavior of the system; nevertheless, it enables to increase designers and developers' trust in the system quality.

Moreover, a test is driven by verification objectives: performance, functional requirement, robustness, *etc.* Each of the test categories is important and provides complementary information on the systems. Among testing objectives, the introduction of dependability raises the issue of the verification of fault tolerance mechanisms. Similarly to other verification techniques mentioned before, it is mandatory to introduce faults or errors in the system to validate the fault tolerance mechanism during the testing phase. This technique called **fault injection** (FI) will be discussed in the following section.

1.3 Fault Injection for the Verification and Validation of Automotive E/E Systems

The introduction of fault injection in ISO 26262, in 2011, has renewed the interest of this method in the automotive industry. However, this well-established verification method is now used by many industries in several domains. Fault injection (Barbosa, Karlsson, Madeira, & Vieira, 2012) is a key technique in the evaluation of the dependability of systems. We have seen in the previous section that fault injection was a dedicated method for both *fault forecasting,* by predicting the post-deployment behavior of the systems under real threats and *fault removal*, by identifying weaknesses or defects in the implementation of safety mechanisms.

In this section, we first identify the objectives of fault injection campaigns, and show how to characterize a fault injection environment, particularly the attributes of fault injection. We also give an overview of typical fault injection techniques and tools, and finally several studies related to the automotive domain are discussed.

1.3.1 Known Approaches

The insertion of faults into systems during the verification phase has been recognized useful in many works to enhance the quality of service regarding fault handling, and thus to improve the dependability of a system. The first approach is based on the idea that the environment of the target is corrupted. Hence, the goal is to evaluate the ability of the system to handle unexpected inputs, caused by a fault in the environment of the system under test. This approach has been investigated in RIFLE (Madeira, Rela, Moreira, & Silva, 1994), BALLISTA project (Koopman, DeVale, & DeVale, 2008).

A second approach consists in performing a modification of the target by inserting an artificial fault and observing the behavior of the reaction of the target. The objective of the latter is the validation of the internal fault tolerance mechanisms or/and the evaluation of the failure modes distribution (Albinet, Arlat, & Fabre, 2004).

Both approaches can be applied to perform robustness testing and dependability benchmarking (DBench, 2004). A framework for defining dependability benchmarks for computer systems was developed in the DBench European project. This framework emphasizes the validation of Commercial-Of-The Shelf (COTS) components, in particular operating systems (*e.g.* several Linux and Windows versions).

Finally, another fault injection approach evaluates the ability of the tests cases to detect faults: *Mutation testing* (DeMillo, Guindi, McCracken, Offutt, & King, 1988). This well-known technique allows the improvement of software quality during the development. In this approach, bugs are introduced in a program: manually – hand-seeded faults –, or automatically generated (mutants) using rules introducing defects. In the following, we will concentrate on safety related, deterministic testing, hence this last fault injection technique will not be developed.

1.3.2 FARM

Several studies have proposed a structure for fault injection environments. To set up a fault injection campaign (Christmansson & Chillarege, 1996), several questions need to be addressed:

1. ***What** is the **appropriate error model** that mimics representative software faults?*
2. ***Where** should **the error** be injected to emulate a particular software fault?*
3. ***When** should **the error** be injected?*
4. ***How** should a **representative operational profile** (i.e. a probabilistic description of system usage) be designed that will maintain reasonable experiment times?*
5. ***What readouts** should be **collected**, and which measures should be calculated?*
6. ***How** should the **calculated measures** be related to analytical models of dependability?*

Among the various fault injection environment models based on the above key issues, one has been particularly used in different works to characterize fault injection on a **T**arget: the **FARM** model (Arlat, et al., 1990), (Arlat, Costes, Crouzet, Laprie, & Powell, 1993), (Benso A., 2011). The FARM model is composed of the four following attributes:

- the set of faults to be injected (the **F**ault model),
- the system activities under which the faults are injected (the **A**ctivation),
- the **R**eadouts of the experiment results,
- and the **M**easures evaluated, based on data of the experiments **< F, A, R >**.

The **FARM** model characterizes in an effective way the fault injection environment, and it is used in this study as a reference for the definition of fault injection experiments.

1.3.2.1 Fault Model

The set of faults to be injected into the target is also called a fault list. Each fault is characterized by a model (*e.g.* stuck-at, bit-flip, *etc.*), a location (*e.g.* memory address, a pin, *etc.*) and an injection time (*e.g.* event-driven, after a given time, *etc.*).

Generally, the size of the fault list is assumed to be infinite. An exhaustive set of experiments covering the full fault list in fault injection campaign is impossible to achieve. In practice, the fault list used to perform the experiments is a subset of the entire fault list that can be injected in a reasonable time but still able to provide significant results: the main criteria here is the **representativeness** of the fault model. Many studies have dealt with this problem (Natella R. , 2011) (Costa, Silva, & Madeira, 2009) (Natella, Cotroneo, Duraes, & Madeira, 2013).

Orthogonal Defect Classification (ODC) (Christmansson & Chillarege, 1996) is a measurement technology that is consistently applied to a large number of IBM projects. The fault types, representing the defects in the source code, are classified in six types, as follows: Assignment, Checking, Algorithm, Timing/Serialization, Interface, and Function.

1.3.2.2 Activation Model

The set of activations **A** specifies how the target is exercised (its functional behavior) during the experiment. It corresponds to a set of functional inputs applied to the target. The complexity of the Activation model directly influences the length of an experiment, as the *injection time* is directly dependent of the **A**ctivation length. This Activation model is often referred as the **W**orkload of the fault injection campaign. An important characteristic of the workload is again its representativeness; ideally, it should be similar to the real behavior of the system in operation. However, most of activation models are implemented with synthetic workload, not representative of the real behavior, but easy to handle and to observe (readouts). An incorrect activation model **A** may result in the two main consequences: *i)* incomplete or non-significant results — the inferred measures obtained on the target are biased; or *ii)* no effect of the faults is perceived — when the fault is not activated by the workload. In this case, the experiment is categorized as harmless whereas it could lead to a critical failure using a different activation set. The Activation model could be defined based on operational profiles to be representative of the activation of the target or scenario-based test from the use cases defined during system definition.

1.3.2.3 Readouts Model

The set of **R**eadouts corresponds to the logged behavior of the system, data and events, execution flow, *etc.* It encompasses all the observations that can be made on the target system. This is strongly dependent on the target system and the fault injection tools. A simulated execution of a system is easier to monitor than a prototype. However, the choice of **R** must be done carefully since it has a strong impact on the results and the analysis. The set of readouts R is composed of variables values, states of the system, detection timing, *etc.*

1.3.2.4 Measures Model

The **M**easures are obtained from the **R**eadouts during or after an experiment. Different types of Measures can be assessed. First, the behavior of the target system in the presence of faults can be evaluated, particularly the failure mode distribution. A severity scale called **CRASH** has been defined in BALLISTA project (Koopman, DeVale, & DeVale, 2008) to characterize the behavior of Operating System and middleware. **CRASH** is an acronym for: **C**atastrophic, **R**estart, **A**bort, **S**ilent, **H**indering. In BALLISTA, a fault corresponds to the corruption of the parameters of a system call executed by a process. The semantics of these failure modes is thus the following:

- Catastrophic: the target computer crashes;
- Restart: the benchmark process hangs and needs a restart;

- Abort: the benchmark process aborts (*e.g. core dump*);
- Silent: no error code is returned when one should have been;
- Hindering: an incorrect error code is generated

It is worth noting a specific scale must be defined for a given target, although some similarities can be found with other existing scales. The definition of these categories is a major issue. Moreover, the measure should not only tackle the first occurrence of a failure mode but also next ones. The work presented in (Albinet, Arlat, & Fabre, 2004) has shown that the first occurrence of a failure is not always sufficient to characterize a fault injection experiment. The detection of the error may occur and in that case, the system is restarted. However, the restart can be insufficient, as another catastrophic failure may occur. Hence, the verification that a reaction has been performed is not sufficient to characterize if the system is safe, and the target should be observed for a second failure.

The main measures concern ***error detection coverage*** (EDC) and ***error recovery coverage*** (ERC). Most of the time these measures are illustrated with pie diagrams distinguishing an error detection sector and another sector giving the distribution of failure modes when the error is not detected. EDC is computed according to Equation 1.1.

$$EDC = \frac{number\ of\ experiments\ with\ detected\ error}{number\ of\ fault\ injection\ experiments} \qquad \text{Equation 1.1}$$

Another important set of measures is the determination of error handling timings, and the Error handling timings of the fault tolerance mechanisms.

The timing requirements of fault tolerance are defined in the ISO 26262 part 1 (ISO 26262, 2011), and Figure 1.3 illustrates the associated terminology.

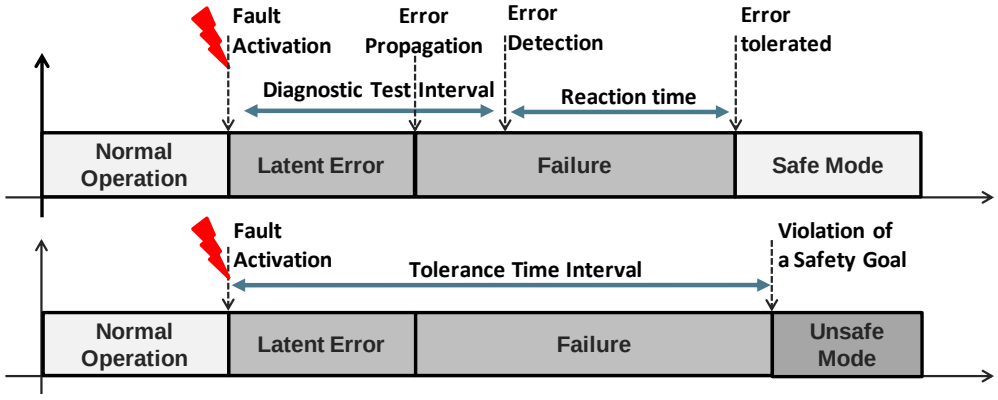


FIGURE 1.3 DIAGNOSTIC TEST INTERVAL, REACTION TIME AND TOLERANCE TIME INTERVAL

First, before the occurrence of the fault, the whole system is in a fault free state or normal operation. Then, when the fault is activated, the system enters in an abnormal system state, in which the error detection mechanisms should handle the error. The error can also remain latent or lead to a failure when there is propagation of the error. When the error is detected, a recovery procedure is needed to handle the error and put the system in a fail-safe state or trigger a degraded mode of operation. The **Diagnostic Test Interval** (***DTI***) is the upper bound of the interval between the occurrence of the fault and the detection, this time is defined according to the period of the recurrent test use to detect the error. The **Reaction Time** (***RT***) is the time between the detection and the end of the recovery.

Finally, a **Tolerance Time Interval** (***TTI***) is the delay between the fault activation and the violation of a safety goal. The Tolerance Time Interval is an intrinsic characteristic of a system. It should be noted

that the transition to a failure mode may not be sufficient, as a failure mode could be tolerated a short amount of time. For example, the loss of the headlights while driving on a motorway is a failure, however, it is considered "safe" if the failure is tolerated within 500 ms.

The TTI is difficult to estimate in practice. However, the following relation must be ensured:

$$DTI + RT < TTI \qquad \text{Equation 1.2}$$

To conclude, it is worth to mention that the **M**easures to be evaluated have to be defined first, since they guide the whole fault injection process. However, their values are evaluated in the last step, by processing information given by the **R**eadouts. The **F**ault model, the system **A**ctivation and the **R**eadouts have to be defined to obtain the significant **M**easures in an efficient way.

1.3.3 Techniques

Several Fault injection techniques have been applied to different types of targets (hardware, software, simulation models, *etc.*). While these techniques are very different in their implementation, they all share the same environment described in the following section. Then, a section is dedicated to an overview the existing fault injection techniques and tools.

1.3.3.1 Environment

A fault injection environment (Hsueh, Tsai, & Iyer, 1997) of a **Target System** should encompass the following components, as defined in Figure 1.4:

- The **Target System.**
- The **Controller** controls the whole experiment, *i.e.*, the **Workload Generator**, the **Fault Injector** and the **Monitor**.
- The **Fault Injector** injects faults selected from the **Fault Library** into the target system.
- The **Workload Generator** generates the inputs, selected from the **Workload Library**, for the target system.
- The **Monitor** tracks the execution of the fault injection experiment for the **Controller** and the **Data Collector**
- The **Data Collector** collects the data (**R**eadouts) during the experiment.
- The **Data Analyzer** analyzes the **R**eadouts collected by the **Data Collector**

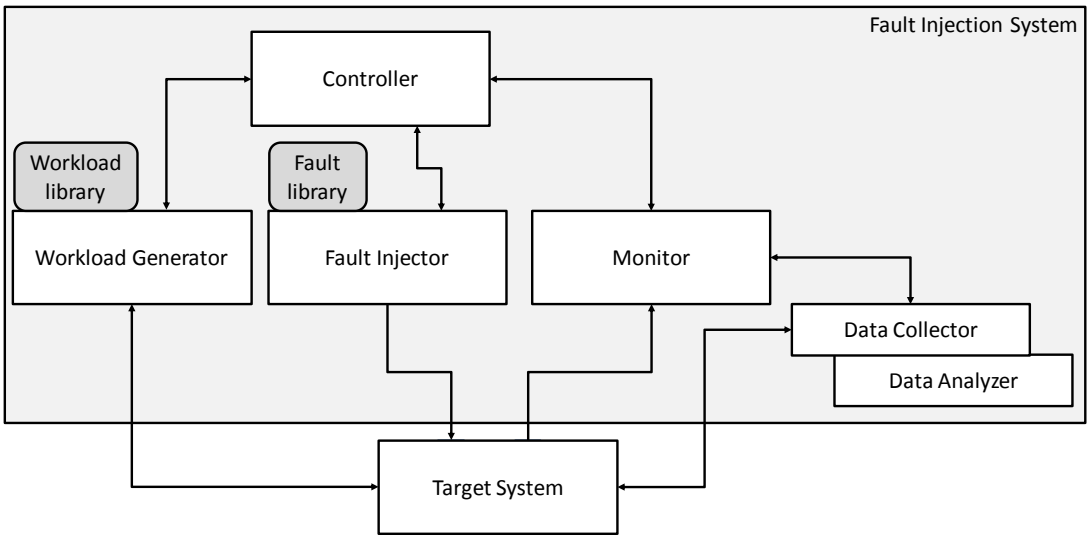


FIGURE 1.4 A TYPICAL FAULT INJECTION ENVIRONMENT (HSUEH, TSAI, & IYER, 1997)

1.3.3.2 Techniques and Tools

Fault injection is a mature technology that has been successfully applied using several techniques on different targets. It is important to notice that the number and the diversity of fault injection techniques is a consequence of the type of targets that have been investigated (hardware, software, models). Many techniques are based on specific tools, often developed for a different purpose (*e.g.* debugging), to perform fault injection. These tools allow either to inject a specific fault model or to control a specific target.

Fault injection techniques can be categorized depending on the target type (Hsueh, Tsai, & Iyer, 1997), (Ziade, Ayoubi, & Velazco, 2004), (Svenningsson R. , 2011). The classification is illustrated in Figure 1.5.

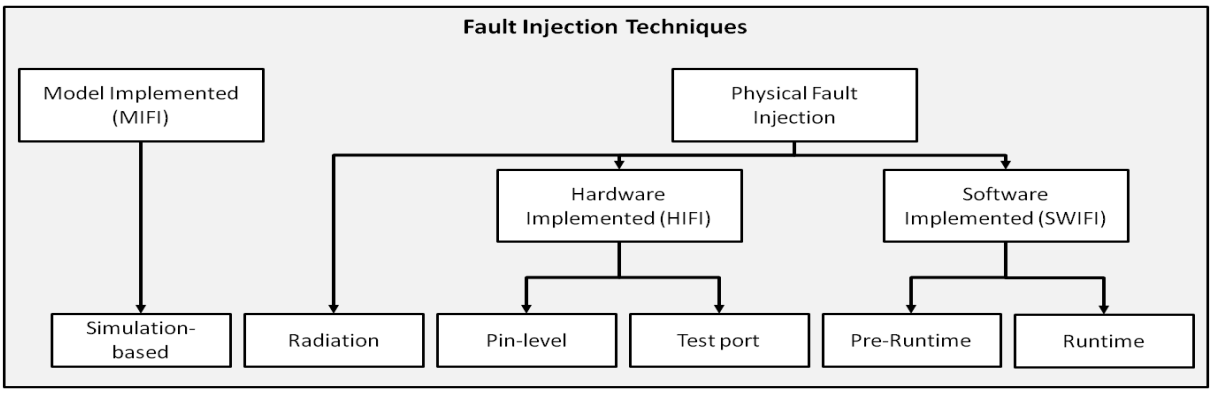


FIGURE 1.5 FAULT INJECTION TECHNIQUES CLASSIFICATION

Physical fault injection can be performed by bombarding the system with either heavy-ions (Karlsson, et al., 1998) or electromagnetic interferences (EMI) (Karlsson, Liden, Dahlgren, Johansson, & Gunneflo, 1994), to mimic Single Event Upset (SEU) that could happen in operation. Today, this technique is less applied as the major drawbacks are the low controllability, and the lack of repeatability of experiments with this method.

Then physical fault injection is divided into two parts:

i) Injection of faults in the hardware of the system (*e.g.* stuck-at faults), *i.e.* Hardware Implemented Fault Injection— HIFI, and

ii) Simulation of physical faults in the software of a system, *i.e.* Software Implemented Fault Injection—SWIFI.

Typically, HIFI corresponds to pin-level fault injection and test-port based fault injection. Pin-level fault injection encompassed techniques that emulate faults by affecting the state of pins of an integrated circuit (Madeira, Rela, Moreira, & Silva, 1994) (Arlat, et al., 1990). Test-port based fault injection techniques rely on debug ports available on several microcontrollers and CPU in order to access the memory of a chip and simulating the effects of hardware faults. One variant of this technique is based on the **Nexus standard** or **IEEE-ISTO 5001-2003** (Nexus5001) (Dees, 2012) that defines a standard debugging interface. This technique has been used in GOOFI (Aidemark, Vinter, Folkesson, & Karlsson, 2001) (Skarin, Barbosa, & Karlsson, 2010) and INERTE (Yuste, de Andrès, Lemus, Serrano, & Gil, 2003). This standard is indeed integrated in the microcontrollers used in the automotive industry.

SWIFI techniques are widely used today, as they are easy to deploy. They can be divided into pre-runtime techniques, *i.e.* the fault is injected in the software before its deployment on the target system (Han, Shin, & Rosenberg, 1995), and runtime techniques. In the latter, the faults are injected during the execution of the software on the target (Benso, et al., 2003) (Kanawati, Kanawati, & Abraham, 1995) (Barbosa, Silva, & Cunha, 2013) (Carreira, Madeira, & Silva, 1998).

Model Implemented Fault Injection has been introduced in (Svenningsson R. , 2011). The injection of faults is performed on models using Simulation-based techniques on VHDL-models of hardware components. This technique was developed when no physical injection solution was manageable on these targets. Hence, the idea was the modeling and the simulation of the fault injection experiments (Jenn E. , Arlat, Rimen, Ohlsson, & Karlsson, 1994) (Jenn E. , Arlat, Rimbn, Ohlsson, & Karlsson, 1994). In recent years, fault injection on model has grown in interest, together with Model Based Development. Indeed, some software modules are not developed in programming language, like C, but with behavioral modeling, *e.g.* Simulink, from which the source code is automatically generated. It is thus possible to test the behavior in presence of fault of the component design when these behavioral models are executed. These techniques have been developed on Simulink (Svenningsson R. , 2011) and SCADE (Vinter, Bromander, Raistrick, & Edler, 2007).

1.3.4 Related Work in Automotive Systems

A quick overview of fault injection has been presented in this chapter. Recently, the introduction of fault injection requirements in the ISO 26262 has renewed the interest of this topic in the automotive industry (Silva, Barbosa, Cunha, & Vieira, 2013) (Rana, et al., 2013). Several subjects have been addressed.

1.3.4.1 Fault Injection in AUTOSAR architecture

First, several studies proposed techniques and tools to perform fault injection in automotive systems, in particular on AUTOSAR-based software architecture. In (Lu, 2009a) (Lu, Fabre, & Kilijian, 2009b) hooks provided by AUTOSAR are used to inject faults in the application but also to monitor its behavior. The same SWIFI technique is used in (Lanigan, Narasimhan, & Fuhrman, 2010), with hooks, to develop a framework based on CANoe from Vector, in order to control the whole experiment through the CAN Network. The approach developed in (Piper, Winter, Manns, & Suri, 2012), is based on the "instrumentation" of the software component of an AUTOSAR application. In this case, the instrumentation is done using a wrapper at the interface between two software components to capture all communication signals. The wrappers allow the implementation of add-on functionalities to control the fault injection experiments.

A similar approach is evaluated in (Islam, Karunakaran, Haraldsson, Bernin, & Karlsson, 2014) (Karunakaran, 2013), where the instrumentation of the wrapper is done at binary level to perform Binary Level Fault Injection (BLFI). This is an intrusive method as the size of the binary integrates the wrapper code. However, the source code of the target application is not modified. In this study, the execution time overhead is quite low but it generally depends on the functionalities added in the wrappers and the number of wrappers. Finally, (Salkham, Pecchia, & Silva, 2013) proposes an approach based on the AUTOSAR's Complex Device Driver (CDD) in order to control the experiment. The objective here is to take advantage of the generic implementation of AUTOSAR to provide a framework in which the controller can be easily implemented in several projects without modifying the basic software.

1.3.4.2 Fault Injection in Simulink Models

Then, as it has already been discussed in the previous section, several works have tackled the injection of faults in Simulink models, as in the MODIFI project (Svenningsson, Eriksson, Vinter, & Törngren, 2010) (Svenningsson R. , 2011) (Svenningsson, Vinter, Eriksson, & Törngren, 2010). Moreover, the tests performed on models have been validated by injection of the same error on a prototype target. (Rana, et al., 2013) also proposed a similar approach to perform fault injection on Simulink models. Again, the choice of Simulink models is driven by their frequent use in the development of application and embedded functions.

1.3.4.3 Fault Injection Experiment definition using Safety Analysis

Few studies have investigated the similarities between safety analysis and fault injection. The main objective is to use the results of safety analyses as FMEA in order to improve the fault injection campaigns. *Yogitech* (Yogitech, 2015) proposed a method (Mariani & Boschi, 2007), (Mariani, Boschi, & Colucci, 2007), (Mariani, Fuhrmann, & Vittorelli, 2006), to perform the verification at System-on-Chip (SoC) level, according to IEC 61508, using FMEA. The approach use FMEA to determine the "sensible zones" in which the faults are injected. Then, the fault injection experiments are simulated, using IEEE *e* standard Verification Language (IEEE Std., 2006) in *Specman* tool from *Cadence*. The main objective is to verify SoC architecture using fault injection.

Moreover, (Bidokhti, 2009) discusses the complementarity between FMEA and fault insertion tests. These tests, hardware implemented fault injection experiments, are performed on hardware parts to improve verification. Fault injection helps to validate FMEA.

1.3.4.4 Other Studies

The work described in (Blin, Laarouchi, & Quéré, 2014) proposes two interesting techniques (one using virtualization and one using emulation) to inject errors in memories without altering the source code of the application. Concerning the emulation techniques, a major advantage is that the entire control of timing aspects of emulated OS, *i.e.*, the temporal impact on behavior of the application, is low. Moreover, emulation allows quick unit testing without deploying the software application. The main drawback of this method is the need of an emulation framework and its cost.

The BeSafe project defines the foundation of functional safety benchmarking of automotive E/E systems. The work presented in (Islam, et al., 2013) tackles the problem of the benchmark targets and the benchmark measures on Safety Elements out of Context – SEooC. SEooC is a concept defined in the part 10 of the ISO 26262. It addresses safety-related elements that are not developed in the context of a particular vehicle but assumptions that have to be validated before integration into the final system. Finally, fault injection has been used in several studies in order to verify automotive products (Trawczynski, Sosnowski, & Gawkowski, 2008), for instance the verification of an Anti Breaking System (ABS). This is a key concept also in the EASIS project, where fault injection has been used to validate fault tolerant architectures. These architectures centralise error detection and error handling in a dedicated software safety mechanisms called Fault Management Framework—FMF (Xi, 2008).

1.4 Conclusion

In this chapter, we have shown that ISO 26262 Standard has motivated new safety practices in the automotive industry. The ISO 26262 redefined specific dependability concepts that should be applied by the actors of the automotive domain.

Fault injection is now a highly recommended method in the ISO 26262, and is required in early phases of the development process. However, we have shown that fault injection has been studied on concrete targets, but to our knowledge, no evaluation of its integration in the development process has been performed yet. The next chapter will describe the development cycle of an automotive embedded system, and the integration of fault injection according to ISO 26262 in this process will be investigated.

Chapter 2 DEVELOPMENT PROCESS & SAFETY

Our objective is to describe a generic development process of Automotive E/E Systems. First, we start by defining the terminology of automotive embedded systems, in compliance with the ISO 26262 norm. Then, the functional development process and the safety development process are detailed. Those form the foundations we will base on for our study, since our final goal is to integrate fault injection in this process. Then, ISO 26262 requirements on fault injection are identified. They represent a second set of constraints our approach must comply with. The final section describes the thesis expected outcomes.

2.1 Development Process of Automotive E/E Embedded Systems

2.1.1 Automotive Embedded Systems

The final objective of the development process in the automotive industry is the production of vehicles. Today, the design of a vehicle is complex as it imposes to integrate a large number of systems to be competitive on the market and to offer the functionalities requested by the customers. There is a wide range of systems: thermal regulation (heater, air conditioning), driving assistance (ABS, ESP…), combustion engine system, electrical systems, Advanced Driver Assistance Systems (ADAS), visibility systems (headlights, wipers), *etc.* All these systems are specified by OEMs, which integrate them in the vehicle.

An E/E system performs functions that drive actuators of mechanical, electromagnetic, hydraulic or chemical nature, according to information gathered by sensors (Human Machine Interfaces (HMI), sensors of a physical quantity (*e.g.*, voltage, current, pressure)).

To develop these systems in a project, the functions are refined at different levels, numbered from (L0) up to (L3), until elementary components are reached. System is the highest level of abstraction (L0), and (L3) deals with elementary components. All these levels are illustrated in Figure 2.1.

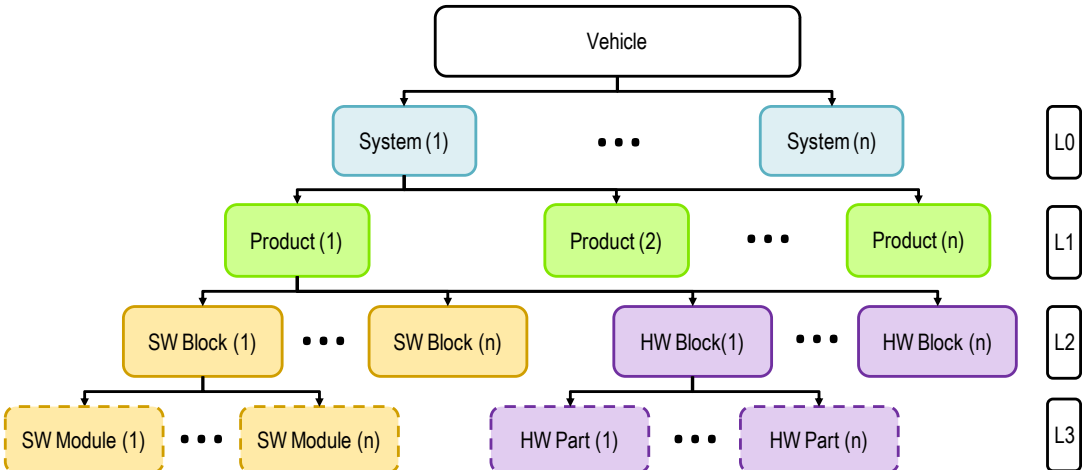


FIGURE 2.1 ARCHITECTURAL ABSTRACTION LEVELS OF A VEHICLE

It should be noted that the terminology used in the thesis is slightly different from the ISO 26262. However, this terminology is completely compatible with the terminology of the ISO 26262, and refines it to comply with the terminology used in Valeo. The differences will be highlighted in the description.

The System (level L0) is decomposed into several Products (L1). The functions of the system are distributed between these products and they share information via a network. Finally, a product is com-

posed of at least a microcontroller that will handle a part of the function of the system, *e.g.*, handle one sensor, drive one actuator, *etc.* This level is not mandatory but most of the systems involve more than one product to achieve a function.

In the ISO 26262 standard, no difference is made between System and Product levels. They correspond to two consecutive *System level* and *Sub-System level*. The main difference with our terminology is that it may exist as many sub-system levels as required. We considered a special case with two levels.

Then, functional requirements of a Product can be refined into Functional Blocks requirements (L2). These requirements are then allocated to hardware, software or both. Then, this allocation defines *Hardware Blocks* and *Software Blocks* (L2). In ISO 26262, they are referred to as *Hardware component level* and *Software component level* respectively.

The *Software Blocks* encompass both the static and the dynamic architectures of the software. The static architecture describes the structure of the software in layers and stacks. It also defines the interfaces between the functions gathered into *Software Modules* (L3). Then, the dynamic architecture describes the configuration of the operating system (tasks priorities and periodicity, interruptions, events, *etc.*), and the mapping of the "function calls" on these tasks.

The designed architecture, populated with implementation of SW Modules, must meet the real-time constraints when executed or programmed on the HW Blocks. In ISO 26262, *Software modules* are referred to as *Software Units.*

The HW Block architecture is the description of HW parts used in order to complete Product requirements. This final terminology is the same than the one used in the ISO 26262 Standard. A HW Block may contain two categories of HW Parts (L3):

- **Integrated circuits**: Microcontrollers, Field-Programmable Gate Array (FPGA), RAM, ROM, *etc.* that support the execution of the SW Blocks.
- **Electronic/electric components**: Transistor, diodes, capacitor, resistor, connector, relay, bus interfaces (*e.g.*, CAN, LIN, FlexRay, and Ethernet), *etc.*

2.1.2 System Engineering

System engineering aims at rationalizing the production of a system and the follow-up of the different phases of the life cycle, by taking into account all activities of system lifecycle in a progressive and methodological approach. The system life cycle refers to the following phases: concept, design, production distribution, maintenance and elimination. System engineering is focused on answering the needs expressed by the client at the beginning of the concept phase.

In this work, we focus on several development phases, which encompass the following activities: definition of functional requirements, definition of architectures, implementation, integration, and testing.

These activities enable us to detail the functional requirements, then to propose technical solutions, and finally to verify that the solution answers the needs. In parallel to these technical solutions, environment issues, cost, planning, project management and maintenance will constrain the development process. Hence, the envisioned process should establish strategies in order to comply with the client needs and expected quality, while ensuring a trade-off between cost and time.

In practice, several methods and models have been proposed for system engineering: waterfall, V-cycle, spiral, incremental, prototype-based development, agile development *etc.* They mainly differ in their phase flow but the types of activities are not different. The choice of the method depends on the system to be built, the company design expertise, *etc.*

A specificity of the automotive industry imposes the coordination of interaction between OEM and Tier-1, Tier-2 and Tier-3 suppliers. A common automotive design process starts with the OEM, which describes and provides the system architecture and the products functional requirements to the Tier-1 that, in turn, may delegate the design of lower level elements (L2 or L3) to a Tier-2, *etc.*

The V-cycle development model described hereafter is widely used in the automotive industry.

2.2 V-Cycle Development Model

The V-cycle development model, as depicted in Figure 2.2, describes the relationship between the different activities of the design phase.

The goal of the development activities in the left hand side of the V is to refine the functional requirements at each level of design (levels are referred to as L0 to L3). They are consistently refined from the highest level, L0 (system) to the lowest one, L3 (hardware parts or software modules).

The right hand side of the V cycle corresponds to the verification and validation activities of the system.

In the context of this work, we use the terms **pre-implementation** and **post-implementation phases**, referring respectively to the left hand side of the V, and the right hand side, since the software modules and hardware parts are implemented only at the end of the pre-implementation phase. Software module's implementation corresponds to coding activities

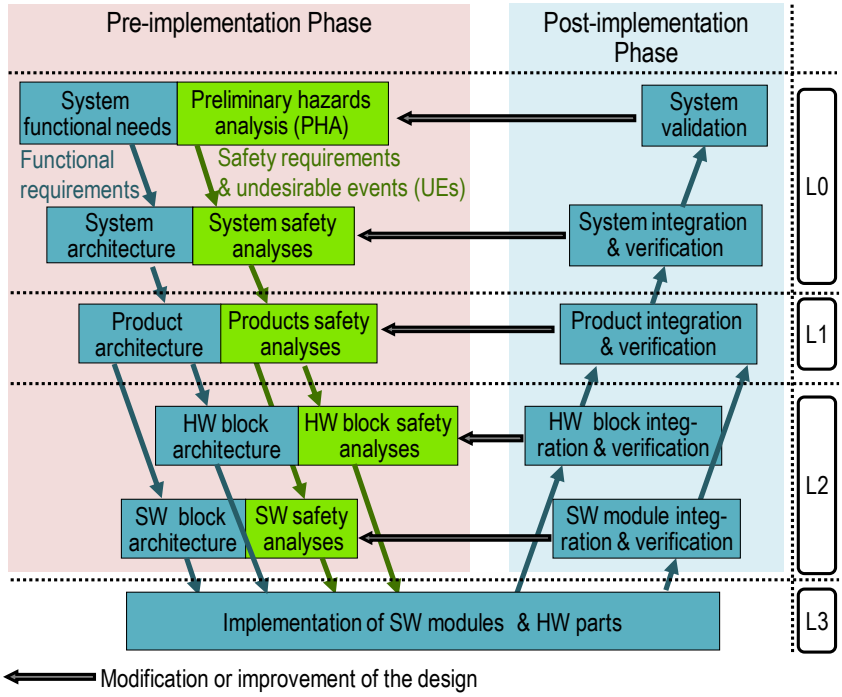


FIGURE 2.2 V-CYCLE DEVELOPMENT PROCESS AND TERMINOLOGY USED

2.2.1 Requirements Analysis

Functional requirements express the services a system must perform, accordingly to the needs defined by the client. The performance of these services (set of functions), *e.g.* timing constraints, the environmental conditions where the system is used, and the eventual operating modes, must be taken into account in the analysis of these requirements.

The functional requirements could be expressed with various forms:

- Textual description: natural language, formal or graphical language;
- Drawings/diagrams:
  - Giving clear definition of the operation of the components involved at a given level. (Static view/ Architectural view);
  - Describing the various interactions and dependencies between these components (Behavioral view/Dynamic view).
- Models of the functions with the following properties:
  - Granularity: brief or detailed;
  - Architectural/behavioral;
  - Executable with software simulation tools

The definition of functional requirements starts with *system functional needs* at L0, which are refined in *system functional requirements*. Then, the system functional requirements are refined in *product functional requirements*, which are, in turn, refined in *hardware blocks functional requirements* and *software Blocks functional requirements* respectively. Finally, the hardware blocks requirements are refined in hardware parts functional requirements and software blocks requirements are refined in Software modules functional requirements.

Today, the system development trend is to use models. Model-based systems engineering (MBSE) tries to formalize modeling for all the activities of the development process, from system requirements to V&V activities. This formalization or abstraction of the real world can be done at all architectural levels. In software design, several languages (Simulink), (Stateflow), (Statemate) are used in the automotive industry; they put forward the development of graphical programming, and hence have been adopted and integrated in current development processes.

2.2.2 Implementation, Integration and Testing Activities

The implementation is a pivot point in the development process. Practically, it consists in the coding activity of SW modules and the integration of HW parts on the E/E circuit. After the implementation, there are two main activities at each architectural level: integration of the considered components of the level, and integration testing. All these tests are part of the verification and validation activities, which are of prime importance in the development process. Indeed, at each level, tests must verify that a given component complies with its specifications. Many tests must be performed at each level, from requirement-based tests to performance tests. The testing process follows a bottom-up approach. It starts with the verification of L3 components, with SW modules unit testing performed on personal computer—PC (SW in-the-loop—SiL) and then executed on the targeted processor (Processor in-the-loop—PiL). HW parts are tested on hardware mock-ups in order to check specific hardware schematics. At L2, SW integration is also tested with PiL, but HW integration is performed on manual test benches.

At L1, testing starts with HW/SW integration tests on HW manual test bench, and often necessitates an emulator or a debugger in order to verify the behavior of the software. Then, the product tests are

performed on automatic test bench. Here, the environment of the product can be simulated. At L0, integration test is performed on the System integration bench, where all products are integrated (the system products are no longer simulated). The final validation is performed by integrating the system into a vehicle.

2.2.3 Relationship between V Branches

The V-cycle model also describes relationships between the two branches of the V (Laprie, et al., 1995). Indeed, at each architectural level, the results of the tests may induce modifications and improvements on design activities. This is the main drawback of this model: as testing is done at the end of the development, the detection of errors in functional requirements or the implementation may significantly affect the development of the project.

Practically, the activities of the V-cycle are not fixed. A system may integrate evolutions in the specifications due to the integration of new functionalities from the client or because of problems identified during a detailed design.

It has been shown that dependability activities must be integrated in the development process, particularly for safety. These activities are introduced at all the steps of the system development process. The safety process is described in the following section.

2.3 Safety Development Process

The complete safety process aims at properly handling functional safety in a project at all architectural levels. The safety process aims at identifying the potential faults leading to a possible hazard and defining the safety concepts, which encompass the specifications of safety requirements (safe state, safety mechanisms independence, ASIL, *etc.*). In Figure 2.2, activities belonging to the safety process are marked in green. Activities of the safety process and their integration in the development process are now described[9].

2.3.1 Safety Analyses at System Level

The safety process begins with the Preliminary Hazard Analysis (PHA) activity that covers the Hazard Analysis and Risk Assessment—HA&RA requirements of the ISO 26262. The main objective of PHA is to identify the system Undesired Events (UE) and to rate them according to their ASIL. Then for each UE, a safety goal (SG), *i.e.* the top-level safety requirements, is defined.

Then, at system level (L0), in parallel to the definition of the System architecture, the following safety activities are performed. The first step is the definition of the *Functional Safety Concept*—FSC. The safety concept formally describes how functional safety will be achieved at the considered level. It refines the safety goals into detailed functional safety requirements that are mapped to the system architecture. These more detailed functional safety requirements correspond to the product safety requirements. The definition of FSC is supported by qualitative and quantitative safety analyses.

[9] For the sake of completeness, Valeo strategy regarding ISO 26262 and the Safety process deployment is described in (Leeman, 2013)

Then, in the automotive safety process, at least one of the two following qualitative analyses is performed: FME(C)A and FTA. These analyses are not specific to the automotive industry and are applied to most development processes when safety is a concern.

2.3.1.1 Failure Mode Effect (and Criticality) Analysis – FME(C)A

FMECA (Bouti & Kadi, 1994), (Department of the Army, 2006), (ECSS-Q-30-02B, 2008) is an inductive approach whose principle is to analyze, for each element (a component or a functional requirement), the consequences of its possible failure modes to identify systematically all the effects on other components and, at the system level, Undesired Events. It can be applied as an accompanying process from the design to the system use phase. In general, the application of an FMECA consists in listing in a table (as exemplified in Table 2.1), based on the functional and structural description of the system, the various failure modes of each component and their characterization. Each failure mode is characterized by:

- (3) Its possible causes.
- (4) The mission phase or a specific operational mode of the element.
- (5-6) Its effect, which can be local, *i.e.*, only the element behavior is affected, or can propagate up to the system level.
- (7) Its criticality.
- (8) The associated detection means and the corrective actions, especially when dealing with a highly critical failure mode.
- (9) The effect in presence of detection and corrective means.

TABLE 2.1 TYPICAL QUALITATIVE FMECA SPREADSHEET LINE

1	2	3	4	5	6	7	8	9
Element	Failure Modes	Potential Causes	Mission Phase/ Operational Mode	Local Effects	Upper-Level Effects	Criticality/ Risk Level	Failure Detection Method /Compensating Provisions	Upper-Level Effect with SM

The criticality of a given failure mode is a categorization of this failure mode based on its severity, its frequency of occurrence, and sometimes, the possibility of detecting earlier symptoms. In automotive systems, it is determined according to the ASIL.

When the criticality of the failure mode is not taken into account, this analysis is referred to as FMEA.

During the operational phase, FME(C)A spreadsheet can be used as a guide for collecting field data for assessing analysis accuracy, aiming at developing maintenance troubleshooting procedures.

However, it is worth noting that the approach has some limitations. For a complex system, it is practically impossible to reach failure modes exhaustiveness. In addition, the approach is not designed to address combinations of failures, since each failure mode is addressed separately. Actually, given the number of failure modes that may be identified, considering their combination raises the problem of combinatorial explosion. Deductive approaches as fault tree analysis intrinsically copes with combination of failures.

2.3.1.2 Fault Tree Analysis

Fault tree analysis (FTA) is a deductive approach, which consists in describing the combinations of events that may lead to a top-level event, which is usually an Undesired Event.

An FTA is based on a graphical representation of the events using logical connectors or gates. Many logical connectors can be found in the literature but the fundamental ones are the AND and OR gates. The resulting diagram, called fault tree, consists in successive levels of events. The top-level event, *i.e.*, the tree root, is the UE. Then, recursively, one determines the causes using a systematic backward-stepping process, until reaching basic events.

FTA is a qualitative analysis activity, but the obtained fault tree may also assist quantitative evaluation.

In Valeo process, FMECA is performed for all ASIL to fulfill the ISO 26262 requirement of an inductive approach. Then, FTA is produced for ASIL C & ASIL D where deductive approach is required by the standard. More generally, FMECA is easy to produce and helps to achieve the exhaustiveness of the analysis of all the potential causes. FTA is also widely used as its graphical formalism is easy to understand and helps identify the critical paths.

2.3.2 Safety Analyses at Product Architecture Level and HW Architectural Level

The qualitative safety analyses at one architectural level could be summarized as follows. Considering the UEs identified at L_{i-1} and the failure modes of the component of the considered level L_i, the goal of safety analyses is the identification of the critical paths between the failures and the UEs. The analysis could be performed in a top-down approach (*e.g.*, FTA) or a bottom-up approach (*e.g.*, FMECA). Finally, the critical path identified enables to determine the UEs of L_{i+1}, the safety requirements and their ASIL. This could be repeated recursively at all following levels, especially at product (L1) and hardware (L2) levels.

2.3.3 Quantitative Safety Analyses

In order to assess the system, product and hardware architecture, ISO 26262 defines three metrics that should be fulfilled. These metrics are calculated based on the **hardware parts' failure rate** (the frequency of a component failure expressed in failures per hour) and the **diagnostic coverage (DC) of the safety mechanisms** (estimation of the coverage of the safety mechanisms). These inputs are given by component suppliers that test intensively samples of their components or by standard tables from historical database of industrial, government or commercial sources. Regarding diagnostic coverage, Annex D of the ISO 26262 standard provides estimations of the achievable coverage depending on the failure modes considered in the safety analyses. Concerning the metrics (**PMHF**, **SPFM** and **LFM**, see Section 1.2.3.2), they must fit within the budgets fixed for each ASIL level and defined in the ISO 26262. More information can be found on the definition and the computation of architectural metrics in (Leeman, 2013), (L'Hostis, 2013), (Cherfi, Leeman, & Rauzy, 2014).

These metrics can be obtained from the qualitative safety analyses by enriching the analysis with numbers. This can be computed from the FMECA, when the failure mode could be associated with a failure rate (for HW parts) and the proposed safety mechanisms could be associated with a DC,

see Table 2.2. The FMECA including the quantitative analyses are often referred to as: Failure Mode, Effect and Diagnosis Analysis (FMEDA), in the automotive industry. Similarly, the same approach is done with FTA. Then, the FMECA tables or the FTA trees enriched with quantitative values are used to compute the different metrics.

TABLE 2.2 EXAMPLE OF ENRICHED FMECA SPREADSHEET WITH QUANTITATIVE DATA

Element	Failure modes	Failure rate	Potential causes	Mission Phase/ Operational Mode	Local effects	Upper-level effects	Criticality / Risk level	Failure detection method / compensating provisions	Diagnostic Coverage (error detection or tolerance coverage)	Upper-level effect with SM

2.3.4 Safety Analyses at Software Architecture Level

Similarly to other levels of architecture, the safety analyses must consider the propagation of failures between software modules and verify that they are mitigated by safety mechanisms. These activities also lead to refine the SW safety requirements and the allocation of ASILs to the SW modules. ISO 26262 explicitly requires the use of specific methods, such as FMECA, FTA and Critical Path Analysis (CPA).

A major aspect in software is the evaluation of the **Freedom From Interference (FFI)** property. Nowadays, the processing capabilities of the microcontrollers used for automotive systems allow designing more complex products. The software design takes advantage of these resources, by proposing an integration of several applications on one microcontroller. In parallel, these systems also embed more critical functions. An important safety issue appears with the integration of applications with different ASIL levels in the same microcontroller. According to ISO 26262, all the modules on a given microcontroller should be developed according to the highest ASIL that apply on the microcontroller, because of the strong interrelationship between these applications. The main drawback is that the application with the lower ASIL is required to be developed with unnecessary efforts. Indeed, higher ASIL modules require applying more complex techniques and methods.

However, the ISO 26262 standard allows the integration of modules with different ASILs, but imposes to prove that the FFI is ensured. FFI is defined as the "absence of cascading failures between two or more elements that could lead to the violation of a safety requirement". As an example, the integration of a Quality Management (QM) or lower ASIL module is allowed if and only if it can be proven that it does not interfere directly or indirectly with the behavior of any higher ASIL co-located modules. The following interferences have to be considered between the two software applications: *i)* corruption of shared-data, *ii)* calls of Application Programming Interface (API) service, *iii)* the real-time behavior (*e.g.*, scheduling, task pre-emption), *iv)* shared-memory access and *v)* shared hardware peripherals. Safety mechanisms may be added to handle such inteferences. This activity is referred to as a FFI Analysis—FFIA.

2.3.5 Safety Tests

Depending on the ASIL allocated to the component under test, it may be necessary to strengthen the existing test strategy to ensure that appropriate combinations of the many test methods required in part 4, part 5 and part 6 of ISO 26262 (ISO 26262, 2011) are used. All safety tests are specified and performed within testing activities, and the test strategy must cover all safety requirements.

In practice, proper testing of the safety mechanisms often requires a stronger involvement of safety engineers. Indeed, robustness tests on a target implementing safety mechanisms are performed with respect to external faults. For instance, incoherent network frames (*e.g.*, CAN, LIN) or out of range

values are sent as inputs to the tested product. However, currently, the injection of arbitrary faults in memory in order to verify software safety mechanisms or the instrumentation of highly integrated hardware is not fully integrated in the process.

2.4 Fault Injection Requirement of ISO 26262

The previous section highlighted that today fault injection is not completely integrated into the development process of automotive systems, yet ISO 26262 requires fault injection all along the development process.

Indeed, the standard highly recommends the use of fault injection techniques throughout the development process, considering both pre- and post-implementations phases, to verify if safety requirements are correctly handled by safety mechanisms.

THE REQUIREMENTS OF THE ISO 26262, WHICH RECOMMEND FAULT INJECTION, ARE RECAPITULATED IN

Table 2.3, and the activities impacted in the development process illustrated in Figure 2.3.

2.4.1 Requirements during Pre-Implementation Phase

TWO REQUIREMENTS PROPOSE FAULT INJECTION DURING PRE-IMPLEMENTATION PHASE. ONE AT SYSTEM LEVEL (***REQUIREMENT 1***) AND ONE AT HARDWARE LEVEL (***REQUIREMENT 8***). THE PRE-IMPLEMENTATION REQUIREMENTS ARE HIGHLIGHTED IN LIGHT GREY IN

Table 2.3. Here, fault injection is a part of simulation-based tests, and aims at "*verifying the safety requirements for compliance and completeness*". We have shown in the Section 1.3.3 that fault injection can be performed on models.

Particularly, fault injection should *check some aspects of the design for which "analytical methods" such as safety analyses are not considered sufficient*. However, the exact objectives of performing fault injection during this phase and the possible links with safety analyses needs to be clarified.

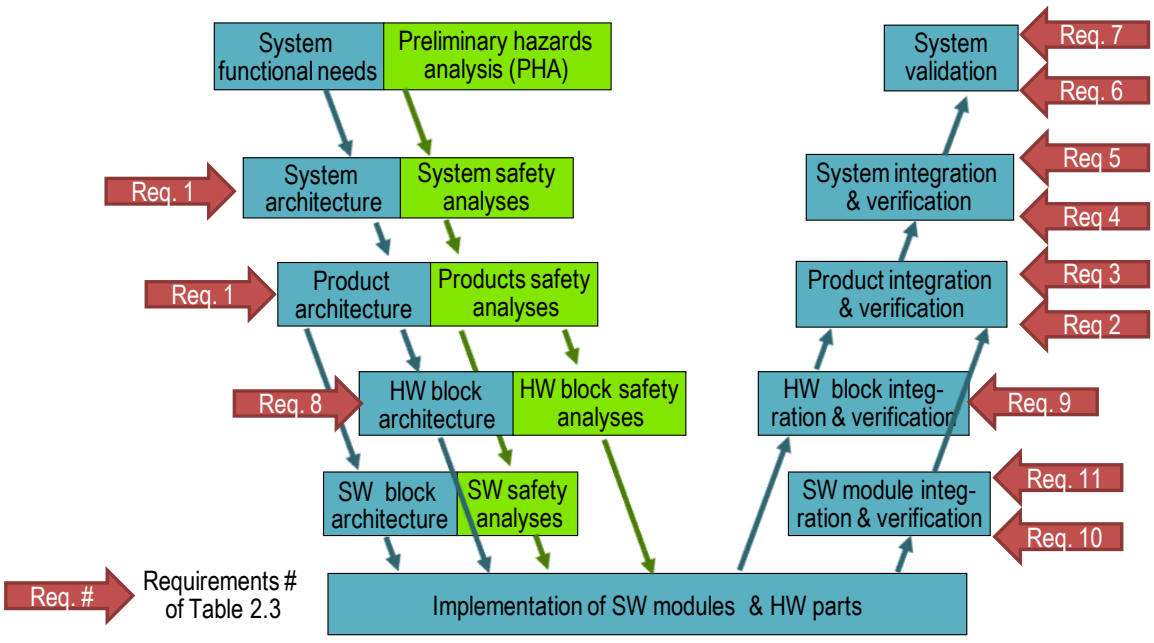


FIGURE 2.3 FAULT INJECTION REQUIREMENTS OF ISO 26262 WITHIN THE DEVELOPMENT PROCESS

TABLE 2.3 ISO 26262 REQUIREMENTS FOR FAULT INJECTION TECHNIQUES

Requirement #	ISO 26262 Chapter		Reference to recommendation	Highly Recommended ASILs (++)
1	Part 4	7.4.8.1 Verification of system design	*The system design shall be verified for compliance and completeness with regard to the technical safety concept using the verification methods.* • Table 3 —System design verification	ASIL C & D
2		8.4.2.2 Hardware-software integration and testing	*The correct implementation of the technical safety requirements at the hardware-software level shall be demonstrated using feasible test methods.* • Table 5 — Correct implementation of technical safety requirements at the hardware-software level	ASIL B, C & D
3			*The effectiveness of the diagnostic coverage of the hardware fault detection mechanisms, with respect to the fault models, shall be ensured by applying feasible test methods.* • Table 8 — Effectiveness of a safety mechanism's diagnostic coverage at the hardware-software level	ASIL C & D
4		8.4.3.2 System integration and testing	*The correct implementation of the functional and technical requirements at the system level shall be demonstrated using feasible test methods.* • Table 10 — Correct implementation of functional safety and technical safety requirements at the system level	ASIL C & D
5			*The effectiveness of the safety mechanisms' failure coverage at the system level shall be demonstrated using feasible test methods.* • Table 13 — Effectiveness of a safety mechanism's failure coverage at the system level	ASIL C & D
6		8.4.4.2 Vehicle integration and testing	*The correct implementation of the functional safety requirements at the vehicle level shall be demonstrated using feasible test methods.* • Table 15 — Correct implementation of the functional safety requirements at the vehicle level	ASIL A, B, C & D
7			*The effectiveness of the safety mechanisms' failure coverage at the vehicle level shall be demonstrated using feasible test methods.* • Table 18 — Effectiveness of a safety mechanism's failure coverage at the vehicle level	ASIL C & D
8	Part 5	7.4.4.1 Verification of the Hardware Design	*The hardware design shall be verified for compliance and completeness with regard to the hardware safety requirements.* • Table 3 —Hardware design verification	N/A
9		10.4.5 Hardware integration and Testing	*The hardware integration and testing activities shall verify the completeness and correctness of the implementation of safety mechanisms with respect to hardware safety requirements.* • Table 11 — Hardware integration tests to verify the completeness and correctness of the safety mechanisms implementation with respect to the hardware safety requirements	ASIL C & D
10	Part 6	9.4.3 Software unit testing	*The software unit testing methods shall be applied to demonstrate that the software units achieve:* • *compliance with the software unit design specification;* • *compliance with the specification of the hardware-software interface* • *the specified functionality;* • *confidence in the absence of unintended functionality;* • *robustness; and* • *sufficient resources to support their functionality.* • Table 10 — Methods for software unit testing	ASIL D
11		10.4.3 Software integration and testing	*The software integration test methods listed in Table 13 shall be applied to demonstrate that both the software components and the embedded software achieve:* • *compliance with the software architectural design;* • *compliance with the specification of the hardware-software interface;* • *the specified functionality;* • *robustness; and* • *sufficient resources to support the functionality.* • Table 13 — Methods for software integration testing	ASIL C & D

2.4.2 Requirements during Post-Implementation Phase

In the post-implementation phase, the objectives of fault injection are well defined. These requirements tackle hardware, software, product, system and vehicle levels. Besides software level, fault injection is a method, which aims at:

1. *demonstrating the effectiveness of the safety mechanisms diagnostic coverage.*

2. *demonstrating the correct implementation of the safety requirements*.

The first objective is to evaluate the efficiency of the implementation, the design or the integration of the safety mechanisms. Fault injection is there defined as a dedicated method, with "error guessing test", to verify these mechanisms. This is only required for high ASILs (ASIL C & ASIL D), and represents a huge amount of work. The entire fault model handled by the safety mechanism should be identified, and the fault injection experiments defined accordingly. Indeed, it is required to estimate the error detection and recovery coverage of the safety mechanisms, and verify that the implementation of the safety mechanism is robust to arbitrary faults or interferences from its environment. This requirement is very demanding, as the fault injection campaign must inject a non-restrictive fault model. However, this is highly recommended for ASIL C & D safety mechanisms. This objective is specified by ***Requirement 3***, ***Requirement 5***, and ***Requirement 7***.

The second objective, *i.e.*, the *correct implementation of safety requirements*, must be verified at different architectural levels, and also applies to components with lower ASILs (depending on the considered level, see

TABLE 2.3).The verification of the absence of violation of safety requirements must be ensured by fault injection, Requirement-based tests or Back-to-back tests. Fault injection helps in the verification of non-occurence of an Undesired Event in the presence of faults. This second objective covers a wider set of critical systems. Indeed, according to ISO 26262, fault injection is required at least for ASIL C and even for ASIL B at HW/SW integration level.

Fault injection must be consistent with other validation activities for all ASILs. Indeed, the **Requirement-based test** (both functional requirements and safety requirements) is another dedicated method to address this objective, but the method is required for all ASILs (from ASIL A to ASIL D). This activity may lead to define fault injection experiments to exercise a safety mechanisms for a given ASIL. This is why fault injection can also be required to ensure that a safety requirement is satisfied at all ASILs.

Contrary to the the first objective, there is no need for injecting an exhaustive fault model. For example, the verification of a safety mechanism should only check its implementation with respect to the failures modes identified in the safety analyses.

This objective is specified by ***Requirement 2***, ***Requirement 4***, ***Requirement 6***, and ***Requirement 9***.

2.4.2.1 SW Unit Testing & SW Integration Testing

At software level, fault injection is a dedicated method for software integration testing and software unit testing, together with ***Requirements-based test***, ***Interface test***, ***Resource usage test***, and ***Back-to-back comparison test between model and code***. The objectives of these methods is to demonstrate that the software module or the software architecture achieves (***Requirement 10***, ***Requirement 11***):

a) compliance with the software unit design specification;
b) compliance with the specification of the hardware-software interface;
c) the specified functionality;
d) confidence in the absence of unintended functionality;
e) robustness, e.g., the absence of inaccessible software / dead code, the effectiveness of error detection and error handling mechanisms ;
f) sufficient resources to support their functionality.

On software, fault injection that *includes the injection of arbitrary faults* (by corrupting values of variables, by introducing code mutation, or by corrupting values of CPU registers) is at least recommended for all ASILs, and is highly recommended for ASIL D and specifically for ASIL C regarding SW Integration testing.

A distinction can be made between the objective of *"compliance with the specifications"*and *"robustness"*. The first one can be associated with the "*demonstration of the correct implementation of the safety requirements*" required at all safety levels by means of Requirement-based tests. Then *"robustness"* should be evaluated for highly critical software units (*i.e.*, ASIL D) or software integration (*i.e.*, ASIL C and D), by the *"demonstration of the effectiveness of the safety mechanisms diagnostic coverage."*

To conclude, at all levels fault injection should be used for all ASILs in order to verify the propagation of the potential causes of failures identified in the safety analyses. On the one hand, fault injection is a dedicated method to verify the correctness of safety analyses. On the other hand, fault injection should also be used in order to verify the robustness of a given component at different levels. Hence, our interpretation of the ISO 26262 standard leads us to apply fault injection tests to all ASILs. Our interpretation is summurized in Table 2.4.

TABLE 2.4 INTERPRETATION OF ISO 26262 REQUIREMENTS FOR FAULT INJECTION TECHNIQUES

Requirement #	Reference to recommendation	Highly Recommended ASILs (++)	Our interpretation of the ISO 26262 (++)
1	*The system design shall be verified for compliance and completeness with regard to the technical safety concept using the verification methods.*	ASIL C & D	
2	*The correct implementation of the technical safety requirements at the hardware-software level shall be demonstrated using feasible test methods.*	ASIL B, C & D	**ASIL A, B, C & D**
3	*The effectiveness of the diagnostic coverage of the hardware fault detection mechanisms, with respect to the fault models, shall be ensured by applying feasible test methods.*	ASIL C & D	
4	*The correct implementation of the functional and technical requirements at the system level shall be demonstrated using feasible test methods.*	ASIL C & D	**ASIL A, B, C & D**
5	*The effectiveness of the safety mechanisms' failure coverage at the system level shall be demonstrated using feasible test methods.*	ASIL C & D	
6	*The correct implementation of the functional safety requirements at the vehicle level shall be demonstrated using feasible test methods.*	ASIL A, B, C & D	
7	*The effectiveness of the safety mechanisms' failure coverage at the vehicle level shall be demonstrated using feasible test methods.*	ASIL C & D	
8	*The hardware design shall be verified for compliance and completeness with regard to the hardware safety requirements.*	N/A	
9	*The hardware integration and testing activities shall verify the completeness and correctness of the implementation of safety mechanisms with respect to hardware safety requirements.*	ASIL C & D	
10 a)	*The software unit testing methods shall be applied to demonstrate that the software units achieve:* *a) compliance with the software unit design specification;* *b) compliance with the specification of the hardware-software interface* *c) the specified functionality;* *d) confidence in the absence of unintended functionality;* *e) sufficient resources to support their functionality.*	ASIL D	**ASIL A, B, C & D**
10 b)	*The software unit testing methods shall be applied to demonstrate that the software units achieve robustness*	ASIL D	
11 a)	*The software integration test methods shall be applied to demonstrate that both the software components and the embedded software achieve:* *compliance with the software architectural design;* *compliance with the specification of the hardware-software interface;* *the specified functionality;* *sufficient resources to support the functionality.*	ASIL C & D	ASIL A, B, C & D
11 b)	*The software integration test methods shall be applied to demonstrate that both the software components and the embedded software achieve robustness*	ASIL C & D	

2.5 Thesis Orientation & Proposed Methodology Overview

In this chapter, the development process of automotive systems has been described and ISO 26262 requirements on fault injection have been discussed.

Firstly, we have shown that fault injection is required during the pre-implementation phase. Several tools have been developed to inject faults or failures into specific models used in various industrial domains, see Section 1.3.3.2 However, these studies do not provide a justification of the interest of fault injection during this phase. There is no continuous process to integrate fault injection at all steps of development cycle. Moreover, as far as we know, only a few works have investigated the integration of fault injection at multiple levels of abstraction, even in the post-implementation phase (Kaâniche, Romano, Kalbarczyk, Iyer, & Karcich, 1998). Hence, we have tried to explore the various facets of fault injection in the different phases.

To reach this goal, we focused on the application of FARM method at each step. FARM is a keystone of our approach, and we consider that fault injection must be based on FARM to be well defined.

In Chapter 3, we will investigate the applicability of FARM during the pre-implementation phase. What are the targets? What are the objectives? Which fault model and activation model should be considered? Then, we will integrate the obtained results into the development process described in this chapter. What are the relationships between fault injection and the requirements? Which links can be drawn with the safety analyses? Finally, the continuum between the different levels of architecture will be explored, by showing how fault injection can be guided between several levels of abstraction.

This contribution will be illustrated on a case study: an Electronic Steering Column Lock (ESCL) System.

We will also tackle, in Chapter 4, the problem of the identification of the fault injection experiments. Following FARM, we define the fault injection experiments during the post-implementation phase. The main problem is the fulfillment of ISO 26262 requirements on fault injection:

1. *the demonstration of the effectiveness of the safety mechanisms.*
2. *the demonstration the correct implementation of the safety requirements.*

Moreover, fault injection campaign may lead to the definition of intensive tests. An objective is to put necessary efforts on the right component. This is why we need to exploit the results of the contribution on pre-implementation phase to prevent unnecessary costly campaigns. However, this must not lead to bias the evaluation of the two previous objectives.

The definition of fault injection experiments during the post implementation phase is important to maintain the traceability of the requirements (functional or safety ones) defined in the pre-implementation phase. This last issue corresponds to the analysis of the fault injection experiments, which can lead to validate, or identify improvements and modifications of the design. In conclusion, in Chapter 4, we will investigate complementarities of fault injection with safety analyses. What are the hypotheses during the pre-implementation (safety analysis) that can be validated by fault injection by experiments?

In Chapter 5, we will apply the overall approach on a case study: a Front-Light System. We will first show using different safety analyses how FIA helps in the definition of the design. We then illustrate the importance of one particular software safety mechanisms in this architecture: the AUTOSAR

Watchdog Manager. We will also show between different levels of abstraction what the critical paths are.

Finally, in Chapter 6, we depict our fault injection environment and give the measures of the experiments resulted from the FIA. These experiments enable to validate the objectives of fault injection in the development process, according to the ISO 26262.

Chapter 3 INTEGRATING FAULT INJECTION IN THE PRE-IMPLEMENTATION PHASE

We have shown that ISO 26262 recommends major efforts for the integration of verification and validation techniques in the safety development process. A particular emphasis should be put on the definition of "state of the art" methods and techniques allocated according to the ASIL of the considered system or entity, and on the improvement of traceability for safety and V&V requirements.

In this chapter, we will present our contribution to fault injection integration during the pre-implementation phase, which is recommended by the ISO 26262 standard, as explained in Chapter 2. Our investigations propose a continuous way to perform safety analyses, during the whole development process.

The chapter ends with an illustration of the method on an Electronic Steering Column Lock (ESCL).

3.1 Is Fault Injection Applicable During the Pre-Implementation Phase?

3.1.1 Preliminaries

We have shown, in Chapter 1, that fault injection covers a large spectrum of techniques, verification and validation activities: from model verification, to software and electrical and electronic devices verification. As Fault Injection is generally used to evaluate implemented targets, we propose to apply a Fault Injection method among those presented in Section 1.3.2 on elements, which exist during the post-implementation phase. In the rest of this section we will explore the meaning of **FARM** on these elements, starting with the post-implementation phase (where Fault injection is commonly used), then considering the pre-implementation phase. The latter constitutes our contribution to the integration of fault injection in the early phases of the development. Figure 3.1 recalls the terminology used in the whole document.

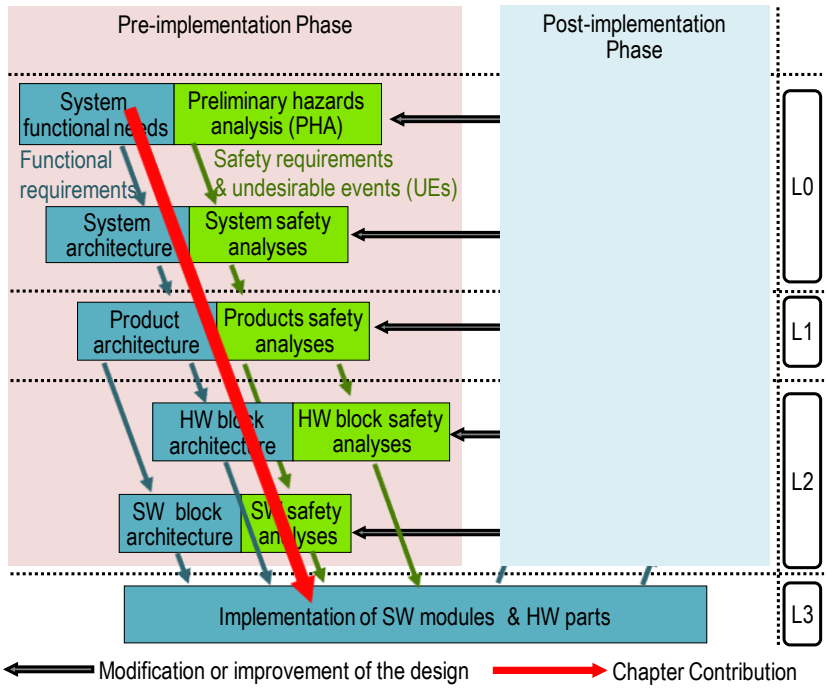


FIGURE 3.1 V-CYCLE DEVELOPMENT PROCESS PHASE ADDRESSED IN CHAPTER 3

Targets

During the post-implementation phase, a fault injection Target will be a set of elements: system / products / HW parts / SW modules that can be accessed/corrupted by using tools. In theory, fault injection experiments may be applied on any target. However, it is usually used to assess the efficiency of the safety mechanisms and the robustness of the architecture with regards to the fault model. Fault injection experiments also aim to verify that fault model does not impact the functions of the target. Hence, Fault Injection target is at least an implementation of functions/actions/tasks, on which measures can be assessed, *e.g.*, the robustness.

During the pre-implementation phase, all existing elements are in the form of functional requirements together with their associated safety requirements and analyses. Fault injection possible targets correspond thus to the ***Functional Requirements*** and all their representations that are produced by a designer/developer along pre-implementation phase, and this, at each considered level. Indeed, these are the only representations that allow us to analyze or compute (when this kind of model is available) the propagation of a fault model.

Hence, we consider that the target of a given level (L_i) is composed of (L_i) functional requirements. For example, at System Level (L_0) we target (L_0) functional requirements: the System functional Requirements.

As the "Targets" are now defined, let us explore the meaning of the FARM method in this context.

Measures

The aim of a measure is to check if a defined safety requirement is handled correctly and to ensure that a safety requirement violation is mitigated as much as possible. These Measures can be either qualitative or quantitative.

- *Qualitative*: characterize the fact whether a specified safety property or a set of properties holds.
- *Quantitative*: correspond to probabilistic or statistical measures on the occurrence of states characterized by property combinations.

During pre-implementation, we cannot define the measures of particular systems or of a given architecture. Particularly, we cannot estimate the distribution of failure modes of the system/element. However, we can identify defense mechanisms that must be evaluated when implemented or we can also identify missing mechanisms that can be added in order to prevent the violation of a safety requirement.

Fault Model

On an implementation, the Fault Model is defined with respect to the Measures of the target to be analyzed. Hence, the fault models are different for distinct measures. A measure can be the distribution of failure modes, or the coverage of a safety mechanism. In addition, the Fault Model heavily depends on the accessibility to the target and on the capabilities of the used fault injection tool.

Hence, at a given architectural level, we can consider:

- failure modes of the *elements of the targets*,
- errors of the *elements of the targets*,
- faults of the *elements of the targets*.

During the pre-implementation phase, *Errors* and *Faults* cannot be defined precisely; they are referred to as potential causes of failure modes. We consider that, in the pre-implementation phase, the potential causes of all failure modes at a given level create the set of faults of the considered level. Moreover, the fault model at a considered abstraction level cannot be more precise than the potential causes that are identified at this level. However, such potential causes can be triggered by low-level faults (*i.e.*, software and hardware faults).

Activation

During post-implementation phase, the fault activation model consists in a set of defined data patterns aimed at exercising the injected faults. This model describes where and when the faults should be injected. It is strongly correlated with the target nature.

This can be specific, or described by representative scenarios in which the fault may be injected at several instants. Depending on this instant, the fault will activate and the error may propagate. These tests cases should be selected in order to mimic scenario that can be encountered during the system lifetime. Additionally, these scenarios have to be chosen carefully to minimize testing complexity.

During the pre-implementation phase, the activation model of an element is related to its functional specification. The definition of the system activation at this step requires the description of the different activation modes, use cases of the target. The more detailed is the modeling of the behavior at the considered level, the more relevant experiment sequences or use cases can be defined.

It is worth to emphasize the primary role of behavioral models during the pre-implementation phase. They allow a thorough understanding of system functions and behavior. In particular, they allow *i)* an easy identification of potential failure causes, and *ii)* a precise analysis of fault propagation.

Readouts

In a fault injection campaign, readouts refer to the observed reactions of the system where a fault F has been injected following an activation model A.

Therefore, during pre-implementation, the readouts are related to the state of the element resulting from the propagation of the injected fault(s). At a given level, they are the effects, which can be analyzed or computed, resulting from the application of the fault model on target assuming an activation model.

Therefore, the failure modes of the considered level are part of the readouts. However, again probability distribution cannot be obtained through experiments in the pre-implementation phase.

3.1.2 Differences between Pre- and Post-Implementation Phases

Performing FI according to the FARM method is meaningful during the pre-implementation phase. Moreover, we also clearly defined the various elements of the method for this phase.

Hence, the main difference between FI during the pre- and post-implementation phases lies in the nature of faults that can be injected, in the control of the fault propagation, and in the measures that can be assessed.

- In the pre-implementation phase, one has to take into account all faults (or at least as much as possible) that may impact safety requirements, in order to analyze their effects and propose

architectural solutions to reduce the effects. During post-implementation phase, all faults may not easily be injected. A detailed analysis might be necessary to group faults into *Equivalent Classes* when they share the same effects. It is a way for optimizing the campaign by reducing the number of FI experiments.

- In the pre-implementation phase, fault propagation must be performed based on assumptions that are applied to the element functional description, either directly or by using executable models or not. In the latter case, model building requires a significant effort, but a tool may help handling the complexity in order to perform a faster analysis. On the other hand, for post-implementation, fault propagation is directly related to the system activity and does not require any specific control.
- In the pre-implementation phase, the measures cannot be estimated or assessed. This is due to the control of the fault propagation and the accuracy of the assumptions regarding the element behavior and fault effects behavior. Therefore, during pre-implementation phase, we can only define the measures that will be later estimated during post–implementation phase.

FI activities related to a system design (description or a model) will be referred to as "**Fault Injection Analysis", FIA**. Conventional FI techniques targeting the real system or a prototype will be referred to as **"Fault Injection by Experimentation", FIE**. With respect to Figure 3.1, FIA corresponds to FI during pre-implementation and FIE to FI during the post-implementation phase.

3.2 Application of the FIA Flow at a Given Architectural Level

We investigate FIA and activities to be performed to produce a well-structured FIA considering the whole pre-implementation phase. These activities are similar at all levels and are illustrated on the product functional requirements level (L_1) considered as an example.

3.2.1 Applying FIA at the Product Level L_1

The different steps of the FIA at the product level are summarized in Figure 3.2 (p.44). This figure also indicates the interactions of this FIA with the upper and lower levels (L_0) and (L_2), as well as the interactions with the safety analyses (for other products at the same level due to fault propagation between products and the system safety analysis).

The outputs of the analyses at the product level correspond to the functional requirements of the HW and SW blocks together with the safety analyses of these blocks.

Definition of the FIA Target at Product Level:

At Product level, the FIA target is a critical product composed of HW & SW Block architecture that follows the functional and the safety requirements. We consider the product as a "grey box".

Definition of the Measures at Product Level:

Fault injection aims to determine if the target handles correctly the effects of the Fault Model.

1) **A first measure concerns the criticality of the effects.** The effect of our Fault Model at Product Level may be a failure mode of the Product, which in turn leads to the violation of a system safety requirement. Hence, the effect of the Fault Model leads to a Product Undesired Events, classified at system level according to its impact on safety. The first measure is the set of analyzed faults that lead to Product UEs. Similarly, we also obtain the set of analyzed faults that do not affect the product safety requirements.

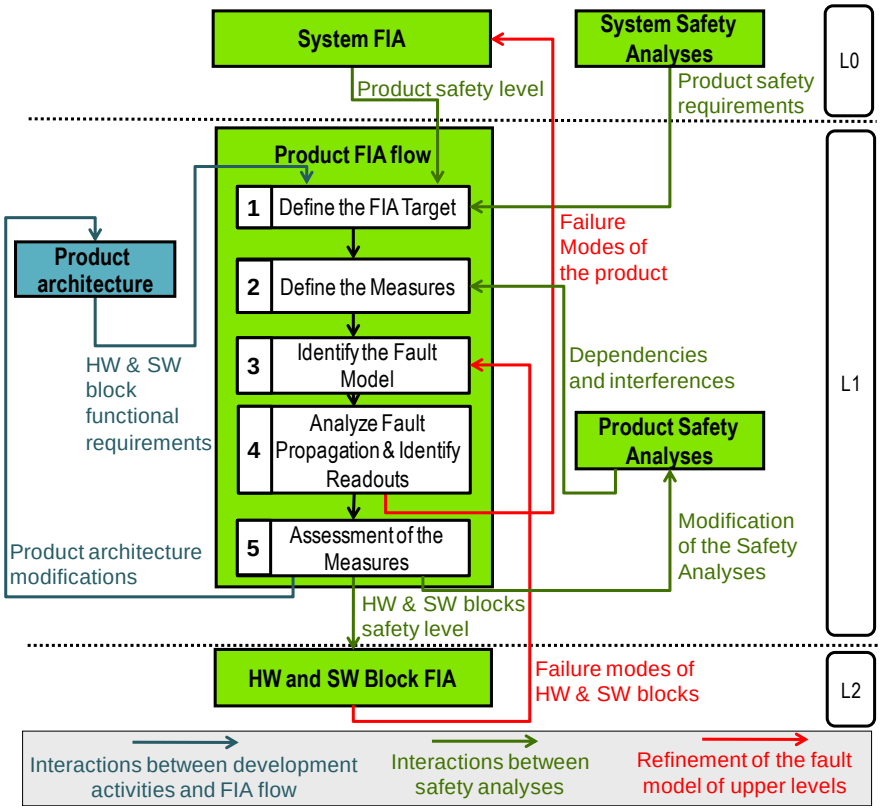


FIGURE 3.2 FIA FLOW OF THE PRODUCT LEVEL AND ITS INTERACTIONS WITH OTHER ACTIVITIES

In addition, it can assess a partial loss of the product, *i.e.*, degradation of functions, safe mode, *etc.*

This measure is linked to the occurrence of associated Product Undesired Event (for quantitative dependability measures). Their distribution will be evaluated during the post-implementation phase based on experiments.

2) **A second measure concerns the mitigations means of the fault model.** When analyzing the propagation of the faults, it can be verified if a safety mechanism is included in order to:
   - Detect possible failure modes (or their causes)
   - Inhibit (or cover) their effects

   During this phase, it is only possible to assess the existence of mechanisms for all failure modes. Their efficiency corresponds to their error detection and recovery coverage (preventing a failure to occur and placing the system in a safe state). In addition, it is not possible to evaluate these numerical values, they only can be evaluated based on experimentation during the post-implementation phase.[10]

At product level, the FIA enables to identify the critical fault model and to propose appropriate safety mechanisms in order to reduce the risk by diminishing the effects and the occurrences of critical failure modes. It is important to note that the measures can be deduced from dependencies and interferences (safety analyses) between the blocks. It is of prime importance at this level with the definition of HW &SW Blocks, which are strongly interlaced. This is an *implementation dependency*.

[10] It should be noted that the considered evaluation does not correspond to the evaluation of the Diagnostic Coverage, which fault model encompasses an estimation of residual faults. However, it is possible to assess the DC based on given fault model and safety mechanisms.

Fault Model Identification at Product Level:
This activity relates to the identification of the possible causes of the product failure modes, *i.e.*, the determination of the possible failure modes of HW & SW block functional requirements. The propagation of these failure modes has to be analyzed. However, the causes of the latter will be completed with information provided by the lower architectural levels (via the link from the Measures of HW and SW block FIA, in Figure 3.2).

The above shows that even though the FIAs should be carried out starting from the highest (system) level down to HW and SW modules, each level needs more detailed information from lower levels to be completed.

Fault Propagation Analysis & Readouts Identification:
Faults are propagated from lower levels to the considered level and from the considered level to upper levels. At product level, the failure modes of the SW & HW Blocks functional Requirements propagate to the product level. Their **effects on the target** should be assessed. These effects are referred to as **local effects** (local with respect to the considered level)

The analysis can be based on the knowledge of the architecture and the behavior defined in the Product Architecture activity. If the definition of the product functional requirements is given by a textual description, the only way to perform this activity is by hand. However, if an executable model exists (model that simulates the propagation of failures) getting the effects can be automated.

Assessment of the Measures:
Based on the knowledge of the Undesired Events at the system level, the analysis of failure modes propagation allows us to attribute a criticality level (or a risk level) to the failure modes of the HW& SW blocks

The FIA enables to determine if there is a defined defense mechanism in order to mitigate the effects of several HW&SW block failure modes. Then, if a mechanism already exists and handles correctly the fault propagation, then the mechanism should be identified and associated with the corresponding failure modes. Otherwise, the product architecture should be modified (as indicated in Figure 3.2), as well as the other safety Analyses of the same product (FMECA/FIA/DFA, *etc.*)

These modifications and improvements, safety mechanisms, have also to be associated with the mitigated HW&SW block failure mode, in order to trace the mechanisms responsible of the mitigation of the fault model.

To conclude this section, at a given level, the FIA flow helps to summarize the results as in Figure 3.3.

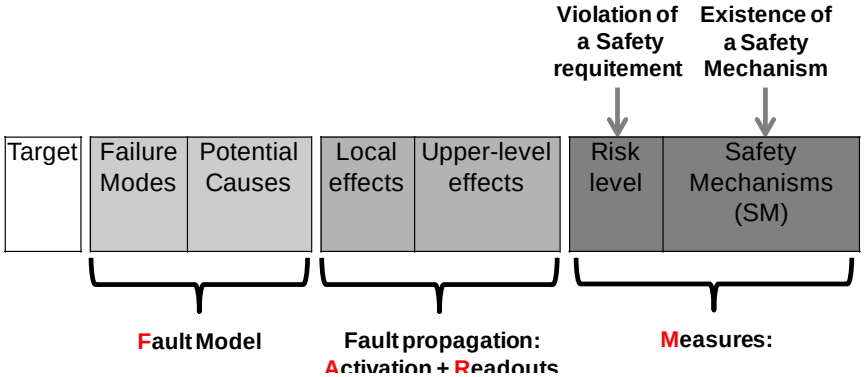


FIGURE 3.3 RESULTS FROM THE FIA FLOW

3.2.2 Relationship between FIA and other Safety Analyses

The various qualitative safety analyses, such as Fault Tree Analyses, Reliability Block Diagrams or cut sets, address the propagation of specific faults at each architectural level and between levels in Section 2.3. Their ultimate aim is the identification of critical paths. This is also the aim of FI during the pre-implementation phase.

More precisely, for a given level, FI and Failure Modes, Effects and (Criticality) Analysis FME(C)A exhibit strong similarities and share several common objectives. An important objective shared between fault injection and FMECA is the identification of all critical faults/failures of the system. This objective is achieved by analyzing the effects of the potential causes of failures on the system in a FMECA, and by analyzing system behavior **in the presence of faults** for the FIA. Both are based on the same kind of analyses. Moreover, both aims at identifying elements that require specific safety mechanisms for error detection or error recovery to mitigate the effects of the critical causes.

From a practical point of view, we can illustrate this analogy by comparing the results of FIA described in the Figure 3.3 with typical information reported in a FMECA spreadsheet, represented in Table 3.1. In this FMECA spreadsheet, an element has to be understood as a function or an entity.

TABLE 3.1 TYPICAL QUALITATIVE FMECA SPREADSHEET LINE (ECSS-Q-30-02B, 2008)

1	2	3	4	5	6	7	8	9
Element	Failure Modes	Potential Causes	Mission Phase/ Operational Mode	Local Effects	Upper-Level Effects	Criticality/ Risk Level	Failure Detection Method/Compensating Provisions	Upper-Level Effect with SM

A FMECA line is guided by the failure modes and their potential causes, supposing the worst activation mode of the system that may propagate the failure mode. However, the activation model is not described in this FMECA line. It is usually provided by the underlying analyses performed to build the FMECA spreadsheet. When necessary, a column "Mission Phase/Operational Mode" (Column 4 of Table 3.1) can be added, in order to precise a static mode of the element, in which the failure mode is considered if the fault propagation is different.

Thus, the activation model and the readouts are represented in the FMECA spreadsheet by:

- Column 4: Mission Phase/Operational Mode
- Column 5: Local Effects
- Column 6: Upper-Level Effects

It is worth to mention that Column 7 in Table 3.1 is related to what is usually called Criticality/Risk level in the FMECA spreadsheet. In the automotive domain, the criticality of the failure modes are usually related to the ASIL **Safety level** introduced in Section 1.2.1.2. These two notions are two facets of the same phenomena.

Column 9 of Table 3.1 is typical in a FMECA. It analyzes the effects on the upper-level, in the presence of the defense mechanisms. According to our analogy, it represents a different FI analysis that is done on the same Element, but with another architecture where the fault activation and propagation is modified, for the same target and fault model.

If two FI analyses are performed on the same target (with without the safety mechanisms) for the same fault model, respectively FIA_i and FIA_{i+1}, the two obtained results show the impact of the added safety mechanisms. This is illustrated in Figure 3.4. Usually the FMECA spreadsheet takes into account these two FIAs, on only one line by aggregating the different propagations and the Measures.

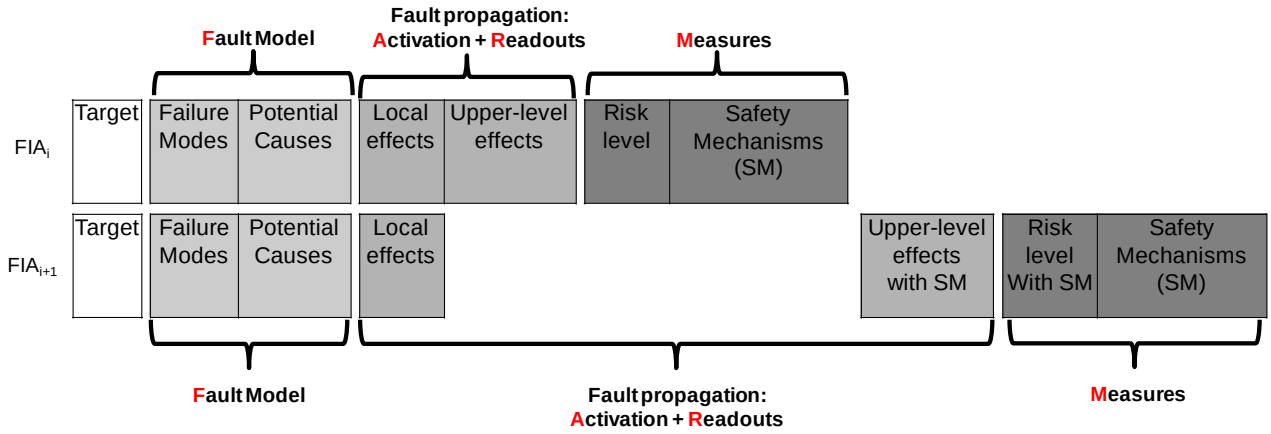


FIGURE 3.4 ITERATION OF FIA FLOW AFTER THE MODIFICATION OF THE ARCHITECTURE

Finally, a set of FI analyses, using as fault model the set of potential causes of failure modes, lead to the identification of the related failure modes of an element in a FMECA. A line of a FMECA spreadsheet can be seen as summarizing the results of a set of FI analyses. In other words, FIA makes visible and explicit the analyses supporting the FMECA spreadsheets.

3.3 Links between FIA Levels

We have introduced the basis of FI analyses at one level during the pre-implementation phase. We have demonstrated the analogy with other safety analyses, particularly FMECA. From now, the FMECA spreadsheet described in Table 3.2 is used to describe the results of the FIA at a defined level.

TABLE 3.2 REPRESENTATIVE FMECA SPREADSHEET

1	2	3	4	5	6	7	8
Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effects	Risk Level	Safety Mechanisms (SM)	Upper-Level Effect with SM

In the following section, the main interest is to link the FIA at various levels in order to link the various analyses and measures.

It is important to note that the links between safety analyses at different architectural levels are well-known. The use of FMECA at multiple levels is already described in various works (ECSS-Q-30-02B, 2008) and is used in projects, particularly in Valeo. However, this is of interest when addressing FIA. Only few works have addressed FI on various levels of abstractions. This is why it is interesting to address what can the results at each level induce on the others. As far as we are aware, hierarchical FI has only been investigated in DEPEND, a simulation-based environment, by injecting faults at several levels of abstraction (Kaâniche, Romano, Kalbarczyk, Iyer, & Karcich, 1998).

3.3.1 S- and Z-shaped Causal Chain

The various levels of the FIA can be linked, based on the propagation of failure modes from one level to the upper level. Two links are of particular interest, based on the following considerations (using the column of Table 3.2):

- link-A: level L_i *Failure modes* (Column 2) corresponds to level L_{i-1} *Upper-level effects* (Column 5).
- link-B: level L_{i-1} *Potential causes* (Column 3) corresponds to level L_i *Failure modes* (Column 2).

These two links are at the origin of two types of causal chains: S-shaped and Z-shaped causal chains.

The S-shaped Causal Chain is based on Link-A (see Figure 3.5). L_i *failure modes* propagate to L_i *upper level effects*, which correspond to a L_{i-1} *failure mode*, which in turn propagates to L_{i-1} *upper-level effect*.

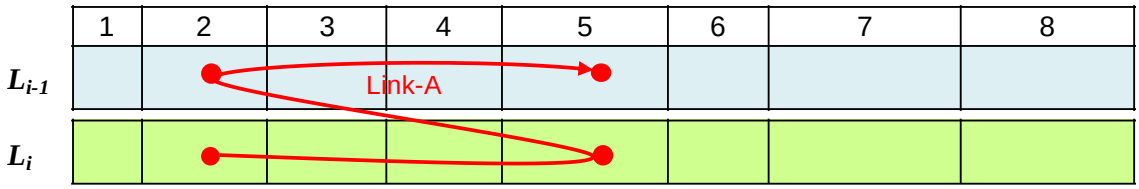


FIGURE 3.5 S-SHAPED CAUSAL CHAIN

An S-shaped chain captures the propagation through the architectural levels of the effects of an initial failure mode. Moreover, the propagation of the initial failure mode (level L_i) of an element E_j may lead to several Upper-level effects. Therefore, L_{i-1} *failure modes* of several elements can be reached. Figure 3.6 highlights this propagation from the element E_j at the level to the element at level L_{i-1}. Then, several elements ($E_1, E_2, \ldots, E_j, \ldots, E_n$) of a considered level can also lead to the same upper-level effect.

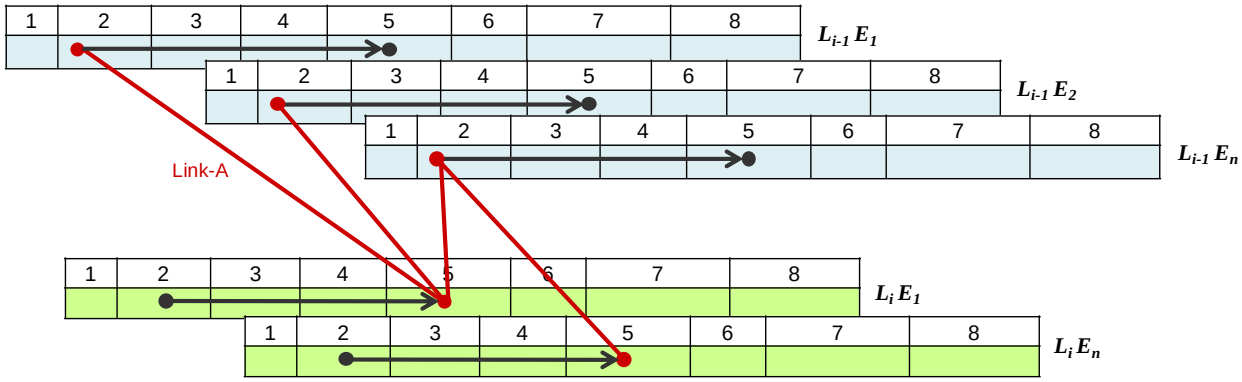


FIGURE 3.6 MULTIPLE S-SHAPED CAUSAL CHAINS FROM AN INITIAL FAILURE MODE

The S-shaped causal chain also provides traceability of the fault model with the safety level defined at the upper-level. This property is due to the assessment of the Upper-level effect, which can be an Undesired Event (associated with a safety level). Hence, the fault model of the considered level is associated with the highest safety level.

This causal chain also enables the definition of safety mechanisms to handle error propagation at the most appropriate architectural levels. It follows the same approach as the Failure Detection, Isolation and Recovery (FDIR) in aerospace systems. This is defined in (NASA, 2005) as "The means to detect off-nominal conditions, isolate the problem to a specific subsystem/entity, and recover of vehicle systems and capabilities. FDIR may be accomplished by the onboard crew, onboard software algorithms, ground commanding, or a combination of the preceding methods." This is a layered approach used to define the different mechanisms (detection, isolation, reconfiguration...) use in order to ensure the dependability of a critical system. In these systems the decision control can be done by ground control, software algorithm or also by hardware protections.

Following one of these chains, it can be determined, what is the best level to handle (detect or recover) the faults in order to mitigate the occurrence of an Undesired Event. For example, adding a safety mechanism at higher level enables to handle a much larger fault model, but it will have a slow reaction time, compared to low-level mechanisms that handles quickly the detected errors but will focus on a dedicated and small fault model. Finally, the decision between several solutions will be done by a tradeoff between advantages and drawbacks of each solution.

This causal chain also helps in the definition of the readouts of FI experiments (FIE) at the successive architectural levels, during the post-implementation phase.

The Z-shaped Causal Chain starts from the potential causes column of level L_i and propagates as a failure mode of the same level, and continues on level L_{i-1} towards the failure mode column. (See Figure 3.7).

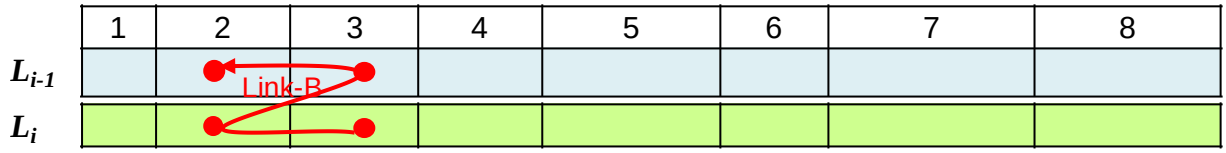


FIGURE 3.7 Z-SHAPED CAUSAL CHAIN

A Z-shaped chain helps refining the fault model of the considered level by identifying equivalent faults, *i.e.*, faults leading to the same failure mode at the upper-level (with the same effects at the system level).

Thus, it will help selecting the fault to be injected during FIE, taking into account the FI instrumentation and the FI accessibility of the target.

Indeed, this chain illustrates the link *"Refinement of the fault model of upper levels"* in the Figure 3.2. Hence, the failure modes of the lower level, will be reported following the identified S-Shaped chain in the column *Potential Causes* of the upper level

Hence, the failure modes of the lower level L_i will be reported following the identified S-Shaped chain in the column *Potential Causes* of the upper level L_{i-1}. Then, it is possible to determine the Z-shaped chains.

Similarly to the S-shaped chain, the Figure 3.8 shows a failure mode of a considered level L_{i-1}, may belong to more than one Z-shaped chains depending on the number of element contributing to the failure mode.

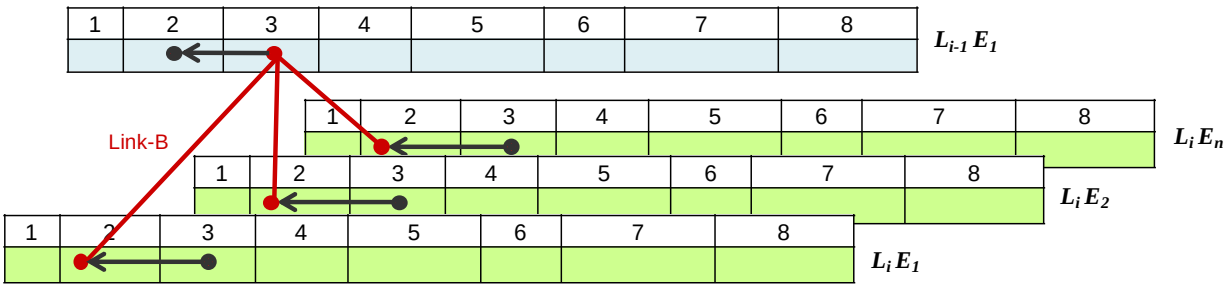


FIGURE 3.8 MULTIPLE Z-SHAPED CAUSAL CHAIN FROM AN INITIAL POTENTIAL CAUSE

3.3.2 Initialization and Termination of the FIA Flow

We described the FIA at a given level by highlighting the goals of each activity and showing how to perform them. We have also shown the links between consecutive levels of FIA, using S- and Z-shaped causal chains. In this section, we describe how the FIA flow can be initialized and terminated, in the framework of the V-cycle of the Figure 3.1.

The FIA flow starts at System level L_0 in which the Definition of the System Architecture has been (or is being) performed. In order to define the measures to be assessed and to assign criticality levels of the various failure modes, *system safety level and system safety requirements* need to be defined respectively in the upper-level FIA and the upper-level safety analyses. In practice, these two activities are part of Hazard and Risk Analyses. In our context, we will use the Preliminary Hazard Analysis (PHA). The PHA aims at identifying undesired events (UEs) and the system safety requirements. The latter are referred to as Safety Goals. Usually, the UEs and the SGs are labeled with the adequate safety level (ASIL according to ISO 26262).

In theory, the FIA flow ends when no targets can be decomposed in sub-elements, or when the lowest fault model granularity is considered. Practically, following our V-cycle, it ends with HW & SW Block level.

At the HW block level, the fault model considered is the set of physical faults that lead to the failure of a HW part. HW parts are the elements that can be handled and for which it can be interesting to assess if a safety mechanism is needed. For example, at this level, the considered failure modes are the short/open circuit of capacitor, resistor, coil, *etc.* or a parameter change of these parts. There is also the stuck-at model for integrated circuits, *e.g.*, inputs or outputs of the circuit stuck at low value or high value.

Going deeper in the fault model will only help a component manufacturer to improve the reliability of the HW part, and this is out of the scope of our work.

At the SW Block level, the lowest fault granularity level is software development/coding faults as they lead to the failure of SW Modules that are the elements of SW architecture.

These faults will be injected on the experimental side of the V-cycle.

3.4 Steering Column Locking System

In this section, we illustrate the application of FIA to a Steering Column Locking System.

3.4.1 System Description

This system controls a locking/unlocking motor on the steering column of the car (see Figure 3.9).

Conventionally steering column locks are purely mechanical and directly coupled to the ignition lock. This decreases the degrees of freedom in the design of a dashboard and of the complete interior. This system is mandatory in the design of vehicle for legal and insurance reasons. Indeed, this is a key system for vehicle theft prevention. The mechanical locking bolt is driven by the Steering Column Locking System. It can be driven to lock the steering column, or by reversing the command, to unlock it.

This is also a frequently used system to illustrate safety in the automotive industry, as it is quite simple, but also as it owns the highest level of criticality (ASIL D).

This system has two functional requirements, which are defined during the "system functional needs" activity:

- The locking management of the steering column when the driver wants to immobilize the vehicle and prevent theft of the vehicle.
- The unlocking management of the steering column when the driver wants to move off.

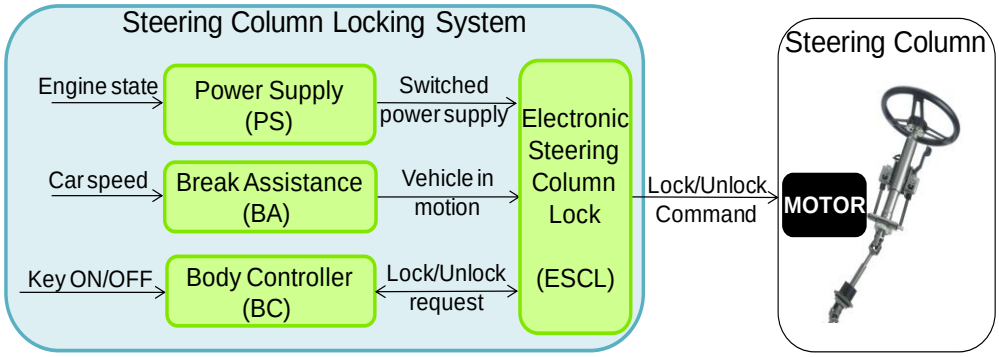


FIGURE 3.9 STEERING COLUMN LOCKING SYSTEM ARCHITECTURE

There are two system safety requirements, referred to as Safety Goals (SG) that must be ensured. These SG are defined in the PHA and each SG is allocated an ASIL:

- **SG1**: The system shall not lock the steering column when the vehicle speed is over a pre-defined threshold. It has the highest safety level, ASIL D.
- **SG2**: If the steering column is locked, the system shall prevent to start the engine of the vehicle. SG2 has the lowest safety level, ASIL A.

In the rest of this section, we will first illustrate the FIA approach at system level L0, to identify failure modes that can violate the safety goals SG1 and SG2, and to check the existence of safety mechanisms preventing their occurrence. Then, we will use results from the FIA at product level (L1) to illustrate the S-shaped chain.

3.4.2 Steering Column Locking System FIA (L0)

FIA Target: *Products Functional Requirements*. The products must ensure the system functional requirements and take into account system safety requirements. The product functional requirements are allocated to the architecture given in Figure 3.9. They are extracted from the System Architecture activity (Figure 3.1) and summarized in Table 3.3. The ESCL is the main product in the system. The three other products provide common functions such as energy, information for safety purpose ("vehicle in motion" information) or centralized information from the driver (the Body Controller collects and transmits commands from the driver' interfaces).

TABLE 3.3 FUNCTIONAL REQUIREMENTS OF THE PRODUCTS

Product	Functional Requirements
ESCL: Electronic Steering Column Locking	ESCL-F1: Lock the Steering column ESCL-F2: Unlock the steering column
BC: Body Controller	BC-F1: Transmit Lock Command from driver's interfaces to ESCL BC-F2: Transmit Unlock Command from driver's interfaces to ESCL
BA: Break Assistance	BA-F: Transmit "vehicle in motion" to the ESCL
PS: Power Supply	PS-F: Supply a switched electrical power to ESCL

Measures: Two main measures can be considered, related respectively to the violation of one of the safety goals SG1 and SG2.

Failure Modes: Table 3.4 lists the failure modes of the ESCL and their local effects. This table is obtained by analyzing functional requirements of ESCL-F1 and ESCL-F2, as well as the propagation of the failures at product level.

TABLE 3.4 FAILURE MODES OF ESCL

Functional Requirement	#	Failure Mode	Product Effects
ESCL-F1	FM1	Spurious Lock	Erroneous lock command
	FM2	ESCL-F1 Lost (No lock)	No lock command is possible
	FM3	ESCL-F1 stuck-at	ESCL always performs lock command
ESCL-F2	FM1	Spurious Unlock	Erroneous unlock command
	FM2	ESCL-F2 Lost(No unlock)	No unlock command is possible
	FM3	ESCL-F2 stuck-at	ESCL always performs unlock command

Fault Propagation and Readouts Identification:

We focus on the fault propagation of the ESCL-F1-FM1: spurious transmission of a lock command when the vehicle is at *high-speed* leads to *Steering column locked* as a local effect while the speed is over the pre-defined threshold. The result at system level is the *locking of the steering column by the ESCL while driving*. The system effect violates the safety goal SG1.

The FIA aims at checking the existence of (or helping the definition of) safety mechanisms to prevent this propagation and the violation of SG1. Two safety mechanisms are identified:

- Braking Assistance product sends vehicle in motion signal to the ESCL when the speed is higher than a defined threshold, thus an ESCL mechanism must check this value before locking the motor. If vehicle in motion is true then SSM1 must inhibit lock command. (SSM1)
- The Power Supply product is a safety mechanism, as it must not power the ESCL when the car engine is running, thus the ESCL is in a safe state. (SSM2)

The FIA of the ESCL-F1-FM1 results in the first line of FMECA in Table 3.5. Similarly, analyzing the other failure modes, we obtain the complete System FMECA table available in APPENDIX 2.

Thus, FIA identifies three safety mechanisms (SSM1, SSM2, and SSM3) whose coverage will be measured through experiments on real target using conventional fault injection. The failure modes will be used to select the most appropriate faults to be injected, *i.e.*, fault that should be handled by each safety mechanisms.

3.4.3 ESCL Product FIA Flow (L1)

To illustrate our approach, we focus on ESCL product. The HW&SW Blocks functional requirements, allocated to the product architecture of Figure 3.10, are:

- Micro-controller Block: it controls the state of the MDB.
- Communication Block: it transmits requests from BC and replies from ESCL.

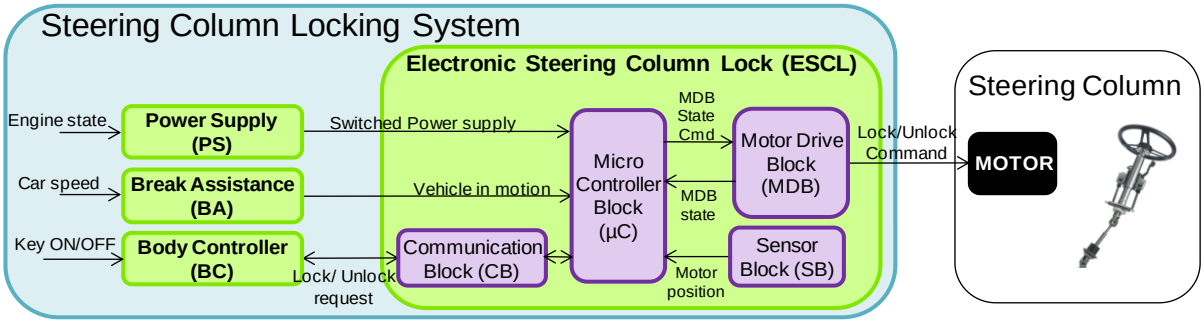


FIGURE 3.10 HW AND SW BLOCKS AT ESCL PRODUCT LEVEL

- Motor Drive Block (MDB): it powers the motor of the steering column. This power converter output is controlled by the micro-controller using four switches: locking, unlocking, breaking the motor and un-power the motor.
- Sensor Block: it senses the position of the motor of the steering column (locked, unlocked, unknown).

Measures: We identified three critical failure modes (or UEs) at system level: Spurious lock (ESCL-F1-FM1), ESCL-F1 stuck-at (ESCL-F1-FM3), No Unlock (ESCL-F2-FM2). At this level, the causes of these UEs should have been identified.

TABLE 3.5 PARTIAL FMECA OF THE STEERING COLUMN LOCKING SYSTEM: ESCL PRODUCT[11]

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety Level	System Safety Mechanisms (SSM)	Upper-Level Effect with SSM
Lock steering column ***ESCL-F1***	Spurious Lock **ESCL-F1-FM1**		Erroneous lock command	Steering column locked while driving **SG1 Violated**	ASIL D	**SSM1:** Vehicle in motion **SSM2:** Switched power supply	No effect *
	ESCL-F1 Lost (No lock) ***ESCL-F1-FM2***		No lock command is possible	Parked vehicle with steering column unlocked		NA	
	ESCL-F1 stuck-at ***ESCL-F1-FM3***		ESCL always performs lock command	Steering column remains locked => vehicle starts with locked column ***SG2 Violated***	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
Unlock steering column ***ESCL-F2***	Spurious Unlock ***ESCL-F2-FM1***		Erroneous unlock command	Parked vehicle with steering column unlocked		NA	
	ESCL-F2 Lost No unlock) ***ESCL-F2-FM2***		No unlock command is possible	Steering column remains locked ==> vehicle starts with locked column ***SG2 Violated***	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
	ESCL-F2 stuck-at ***ESCL-F2-FM3***		ESCL always performs unlock command	Parked vehicle with steering column unlocked		NA	

* assuming a perfect coverage of safety mechanisms

[11] The complete table is available in APPENDIX 2

In this section, we focus on one critical failure mode: "Micro-Controller-F1-FM1, *Erroneous assignment of the outputs of the micro-controller* delivered to the MDB" given in Table 3.6.

The Micro-Controller-F1-FM1 failure mode puts the MDB in a locking state, the MDB powering up the motor in locking mode. In this case, the ESCL triggers a spurious lock of the steering column (ESCL-F1-FM1 failure mode). Two mechanisms are proposed. The first one – PSM1 – is a hardware watchdog which enables the detection of abnormal software behavior. This could be a cause of a spurious locking command sent by the ESCL. Then, a fault tolerance mechanism is implemented to handle single point failure; two independent software modules should be responsible for the locking command of the MDB.

TABLE 3.6 PARTIAL FMECA OF THE ESCL (FAILURE MODE OF THE MICRO-CONTROLLER BLOCK)[12]

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety Level	Safety Mechanisms (SM)	Product Effect with SM
Control the state of the MDB ***µC-F1***	Erroneous assignment of outputs of the micro-controller ***µC-F1-FM1***		Spurious activation of the MDB locking state.	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM1:** Watchdog (HW), **PSM2:** 2 different SW modules should be implemented to control the µC-F1 (redundancy)	No effect*

* assuming a perfect coverage of safety mechanisms

This example illustrates the links between the product and system levels of the architecture, thanks to the S-shaped chain whose elements are indicated in red in Table 3.6 and then in Table 3.5. For illustration purpose, during the pre-implementation phase, this chain helps in the following two activities:

- **Propagation through the architectural levels of the effects of an initial failure mode and traceability of the fault model with the safety level.**

  The failure mode Micro-Controller-F1-FM1 leads to the ESCL-F1-FM1 product failure mode that affects SG1 at system level.
- **Definition of safety mechanisms to handle error propagation at the most appropriate architectural levels.**

  The proposed design includes safety mechanisms (SM) at two levels. PSM1 detects and recovers at product level Micro-Controller-F1-FM1 failure mode. However, if PSM1 fails, ESCL-F1-FM1 occurs but can be covered by SSM1 and SSM2 at system level. Indeed, the lack of coverage of SM1 can be handled by the System safety mechanisms. Both safety mechanisms placed at two different levels are motivated by the required high safety level.

3.5 Synthesis on Fault Injection Analyses

We have demonstrated the continuum in the validation process from fault injection point of view. In addition, the methodology has been illustrated by the use of S and Z-shaped causal chains. We have highlighted that the first one helps in the definition of the experimental measures and the other enables to define the fault model for fault injection experiments. We have shown that FI analyses during the system pre-implementation phase provide information that can be synthesized in FMECA spreadsheets. However, as the critical path are clearly described with the fault model, the possible critical consequences and the fault tolerance mechanisms, it is possible to describe a whole set of attributes of fault injection. FTA or other safety analyses can help to answer fault injection requirements. The ad-

[12] The complete table is available in APPENDIX 3

vantage of FIA is to make visible and explicit all the detailed analyses performed manually or based on models, and tools for activation and fault propagation analyzes.

It should be noted that, even if we have shown that FMECA are the most suitable analyses to answer fault injection objectives, several actors of the industry are used to base their safety processes on FTA rather than on FMECA.

From a practical point of view, FTA is a deductive approach whereas FIA and FMECA are inductive approaches. They can be viewed as complementary. FTA gives an overview of fault propagation not only between the levels but also between elements of the same level. They do not necessarily show explicitly the details related to fault propagation and criticality. On the other hand, FIA and FMECA give more details fault by fault, without showing all the relationships (or propagation) explicitly in the lines. We proposed to analyze the S- and Z-shaped causal chains to provide such a view.

Fault trees representation is interesting for the representation of fault injection. First, the two causal chains, S- and Z-shaped, are represented. Hence, the traceability between levels is shown. The S-shaped correspond to bottom-up reading of a branch of the fault tree. However, the multiple effects of a fault cannot be identified directly, on the fault tree, as a branch only represents a critical path. The Z-shaped chain corresponds to the causes analyzed in the fault tree. S- and Z-shaped chains are respectively illustrated in red and blue in Figure 3.11, in both a fault tree and a FMECA spreadsheet.

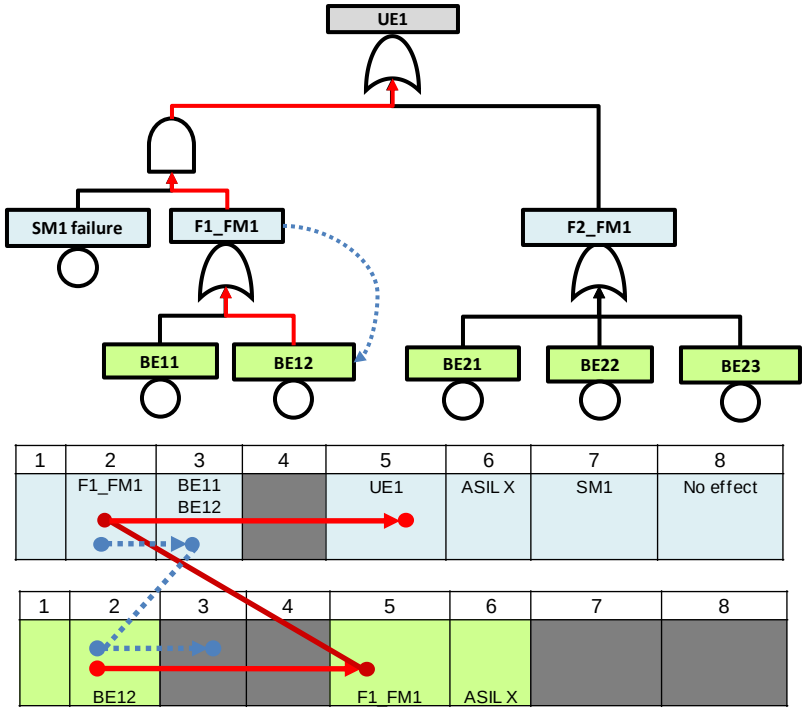


FIGURE 3.11 S- AND Z-SHAPED CAUSAL CHAINS IN FTA AND A FMECA TABLE

Another interesting point concerning FTA is its management of the combination of faults. For the moment, we have only highlighted the effect of single failure, but fault injection is not reserved to single failures. The injection of fault in multiple locations have already been studied, *e.g.*, dual-point fault in (Ayatolahi, Sangchoolie, Johansson, & Karlsson, 2013). Most of the time, these aspects are not tackled by FMECA, therefore FTA can enrich the set of experiments, by adding combination of two or more faults.

In practice, the integration of these experiments in the campaign has to be analyzed. In most of the cases, multiple failures are not exhaustively considered in the analyses, even for dual point faults. A particular attention can be brought to the so-called "second order mechanism"[13]. In the automotive industry, they detect the error of the first order safety mechanisms. In this case, performing a simple fault injection is sufficient to evaluate these mechanisms.

Finally, in order to conform to the ISO 26262 standard, many actors in the automotive domain have started looking for new approaches to FI in the early development phase, with the fear that integrating FI in the early development process will incur a redefinition of the whole development process. Interestingly, the analogy between FIA and safety analyses developed in this chapter shows that FI can indeed be easily integrated in the existing process, and will even improve the efficiency of the global process.

- Safety analyses, FMECA, must be integrated in the development process and, in practice, FMECA is already integrated.
- Safety analyses are required by the ISO 26262 at all levels described before: system, product, software and hardware. Moreover, they are required for all ASILs, even the lowest one.

In the next chapter, we will complete our approach with the determination of the fault injection experiments based on the FIA. Our approach follows studies that have highlights the relationship between FMECA and fault injection experiments.

[13] More information about the characterization of the first and second order safety mechanisms can be founded in (Cherfi, Rauzy, & Leeman, 2014)

Chapter 4 FAULT INJECTION DURING POST-IMPLEMENTATION PHASE

The objective of the chapter is the identification of fault injection experiments on concrete targets. The Readouts and the Measures can be deduced from the S-shaped causal chains. Then, the Z-shaped causal chains enable the identification of the Fault model. Finally, the Activation of the target corresponds to the functional behavior of the targeted element. A fault injection campaign can be defined as a whole for a target belonging to a particular level. Finally, we evaluate the benefits on the FIA of the obtained measures of the FIE.

4.1 FIE Overview

The FIA enabled identifying the propagation of failure modes at different levels of architecture and enabled defining the means (safety mechanisms) to mitigate the propagation at the most appropriate level.

Our aim in the following section is to answer the following question: ***To what extent FIA is of interest for conducting the FI experiments in real targets?***

Fault Injection Experiments aim at checking, during the post-implementation phase, that the various fault tolerance mechanisms defined during the pre-implementation phase are correctly implemented. Another output of such experiments is the quantitative assessment of the most impacting safety parameters.

Firstly, we show how to use the results of pre-implementation phase (Chapter 3) to define the fault injection experiments on targets. Particularly, S- and Z-shaped causal chains will be used to identify the measures and the experiments.

Secondly, we investigate the continuum of the fault injection experiments between the different levels, and we compare obtained measures with FIA. The objective is to validate the measures of the FIA: identification of safety mechanisms and critical paths, with the results of the experiments. In addition, the campaign can be optimized, guided by the FIAs and the causal chains, in order to select efficient experiments. Then, we investigate the complementarity of experiments with analysis.

Finally, we discuss the process and the outcomes with the requirements of the ISO 26262.

The contributions tackled in this chapter with FIE are highlighted in Figure 4.1.

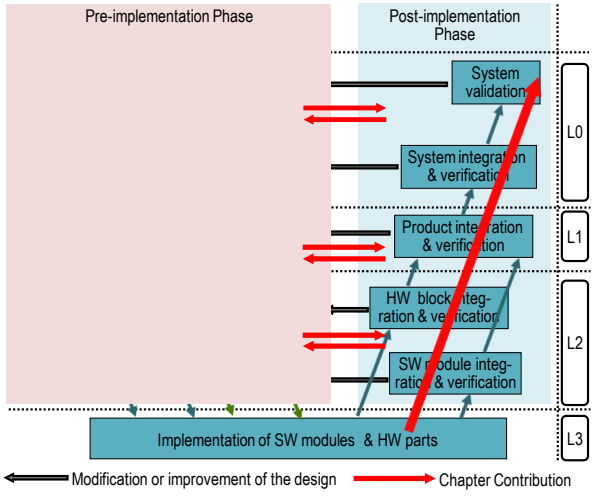


FIGURE 4.1 CONTRIBUTIONS OF CHAPTER 4

4.2 From FIA to FIE: Definition of the Experiments

In this section, the applicability of the FARM method for the identification of the attributes of fault injection experiments and their planning is discussed.

4.2.1 Application of FARM

4.2.1.1 Definition of the Experimentation Targets

The FIA enables the identification of the most critical elements and propagation paths of the architecture thanks to the traceability of ASILs of all the elements, and following the S-shaped causal chain. These critical elements and paths should be particularly verified, in order to validate that the causes of safety requirement violation have been properly mitigated. Considering a specific level of integration, the targets chosen for the experiments have a decreasing ASIL.

The targets usually implement, at least, one safety mechanism (detection or tolerance).Fault injection experiments must demonstrate the efficiency of safety mechanisms with respect to the considered fault model.

4.2.1.2 Measures to be Assessed

Fault injection has two main objectives. On the one hand, FI verifies that the safety requirements are not violated, *i.e.*, the considered error (failures) identified in the safety analyses does not propagate through critical paths to violate safety requirements. The violation of safety requirements can be quantified by identifying the failure modes distribution, in which the failure modes are associated with safety requirements. The failure modes distribution can be represented using “pie chart”, “bar graphs” or “histograms”. In addition, the temporal behavior of error handling is important. The detection time and the reaction time may be part of the definition of the assertion providing the measures. Indeed safety requirements impose a detection or reaction time, to evaluate whether the system handles safely the fault model. Experiments where there is a detection of an error, and experiments where the detection is within the timing requirements should be distinguished. The former are not safe if they do not comply with the timing margins on the contrary to the latter. Finally, these measures enable the evaluation of fault proportion that lead to violate a safety requirement.

On the other hand, FI addresses the verification of the effectiveness of the safety mechanisms, *i.e.*, whether the fault model identified in safety analyses is mitigated by the safety mechanisms. This objective aims at assessing the Error Detection Coverage and / or the Error Recovery Coverage of the safety mechanism. Generally, these results are represented as “pie chart”, in order to illustrate the difference between the faults correctly handled and the coverage deficiency.

These measures quantify the effectiveness of the safety mechanisms preventing the occurrence of undesired events.

4.2.1.3 Faults to be injected

Fault model is extracted from the failure modes of the elements identified for each element or path. If the failure mode is too abstract, it will be refined using the Z-shaped causal chain. The potential causes of the failure modes help identifying elements where the faults/errors can be injected. Two FI strategies are possible, related to two complementary objectives:

- **Objective 1:** verification that a given failure mode identified by the FIA is handled correctly by the implemented safety mechanisms. By injecting representative causes of failures leading to the considered failure mode, the identified safety requirements will be solicited and by doing so, will be easily tested. This strategy relies on the assumption that the failure modes of the element have been deeply analyzed (*e.g.*, important background, former studies).

Figure 4.2 illustrates this strategy. We consider that there is "n" Equivalence Classes (EC) of the failure modes of the SW Module. At least one potential cause identified in the FIA is chosen for each EC. Hence, this enables verifying that the safety mechanism mitigates correctly the Failure Mode of the SW Module or identified a lack of coverage.

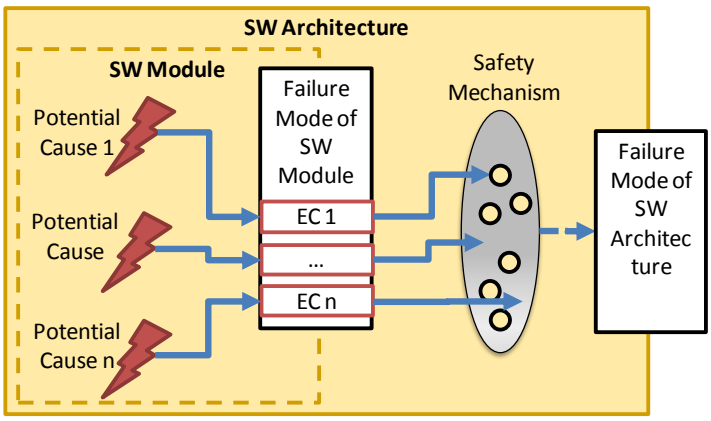


FIGURE 4.2 ILLUSTRATION OF FIRST STRATEGY
(Definition of Fault Model with "n" Potential Causes Representative of the "n" Equivalence Classes)

- **Objective 2:** verification that a set of identified causes of a failure mode will lead effectively to this failure mode. In this case, experiments consist in injecting as much causes as possible to check that the fault/failure propagation paths identified by FIA are valid. In this second strategy, the objective is more exhaustive. This strategy should be applied when the failure mode distribution of the potential causes is difficult to analyze.

Figure 4.3 illustrates this strategy. Here, the safety analyses lead to identify a set of potential causes. All the potential causes should be injected to validate that they are real causes of the failure of the SW Module. When a potential cause leads to a failure mode, the experiment checks whether the propagation is well mitigated.

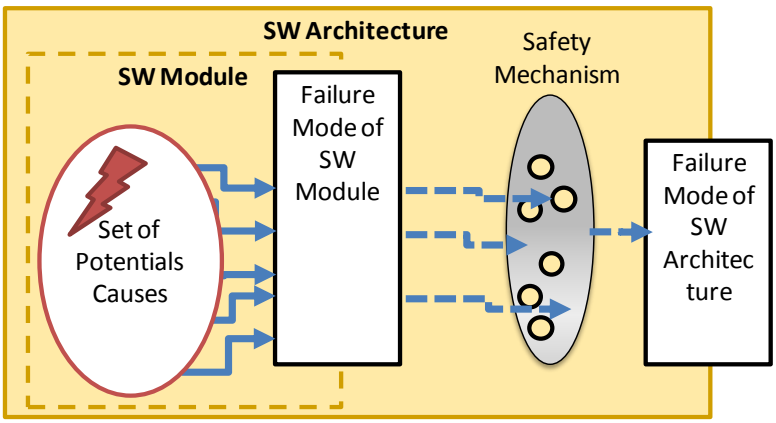


FIGURE 4.3 ILLUSTRATION OF SECOND STRATEGY FOR THE DEFINITION OF FAULT MODEL

At this stage (definition of the fault model), we assume that the strategy is selected independently from FI capabilities provided by the available FI tool. The goal is to define the set of faults that should be injected in order to obtain the desired measures.

4.2.1.4 Activation Model

In the case of FIE, the Activation model is a set of data patterns that aim sat exercising the injected fault. For a general purpose, a solution consists in using a representative program. It aims at evaluating the behavior of the systems in presence of faults during the representative uses of the target. They could be chosen according to the frequency, the criticality, *etc.* of the program.

For the FIE, the Activation model is a set of input patterns that aims at exercising *(i)* the target Element (EUT, Element Under Test) and *(ii)* the location of the injected fault to limit the number of insignificant experiments. The best solution consists in using a representative program that aims at stimulating the EUT in a realistic fashion and at evaluating its behavior in presence of faults. Activation profiles can be selected according to criteria, *e.g.* the frequency, the criticality, *etc.*

Nonetheless, the determination of the Activation set could be significantly improved using behavioral models of the dynamic behavior of the target in its environment. Our proposal is to use the behavioral models developed during the FIA (sequence diagrams, timing diagrams or use cases). The goal is to determine when to trigger the fault injection *i)* from the states of the target's components and/or *ii)* according to specific inputs from the environment.

4.2.1.5 Readouts

The readouts are obtained in order to verify and validate the system according to the safety requirements and to verify the effectiveness of the safety mechanisms, *i.e.*, computation of relevant measures.

It is worth noting that the identification of the readouts should help in the identification of precise ***oracle(s)*** of fault injection experiments. The *oracle problem* (Gaudel, 1995) is one of the main challenges for testers. Observing the tests outputs and deciding whether or not, verification conditions are met is sometime difficult. In fault injection campaigns, there are three main forms of oracles: *i)* specification, *ii)* error detection mechanisms and *iii)* golden run (Leeke & Jhumka, 2009). The specification and error detection mechanisms are related to the definition of a *property* that should be enforced during the experiment. These properties can then be formalized using observation points of the targets, *i.e.* the Readouts. More information on the formalization of robustness tests can be found in (Chu, 2011). The *golden run* is an execution of the system under normal conditions that generates a reference run and its corresponding outputs. The outputs of reference run can be then compared to those of the run when a fault has been injected. However, the golden run approach is recommended for *black box testing*, but in our case, our measures are based on safety requirements evaluation.

To define the readouts, we recommend using the FIA. First, the fault model must be monitored as well as the propagation of the faults (the local effects and higher levels effects). Second, the behavior of the safety mechanisms must be monitored.

Data, variables (physical or digital) and events have to be observed and acquired on the target. Further details about the measurement points (state or events *e.g.*, timestamps, log files of variables, *etc.*) should also be registered. This couple of variables/states indicates whether *i)* a safety mechanism has been triggered and *ii)* the error handling is correct.

Then, from the readouts, a logical expression or a safety property based on the target states can be established to detect when a safety requirement is violated.

4.2.1.6 Assessment of the Measures

The analysis of the readouts leads to assess the measures defined at the beginning of the process. However, as we are performing tests, the results could be ambiguous and must be analyzed. We choose to classify these experiments into four categories, see Table 4.1. We considered experiments that implement at least one safety mechanism, and an experiment on which we verify that safety requirements are ensured. This table ignores faults that have not been activated and that usually fall into the "no observation" category.

TABLE 4.1 READOUTS ANALYSIS

Case	α	β	γ	δ
Activation of a safety mechanism	YES	YES	NO	NO
Safety requirement violated	NO	YES	YES	NO
Comments and further analysis	*Expected results*	*Coverage deficiency*	*Default of the activation of the safety mechanism(s)*	*Fault injected correctly but no effects observed*

The expected behavior – Case α – is the activation of safety mechanisms in the presence of faults preventing the violation of the safety requirement. This behavior should have the highest probability. Obviously, in all other cases, safety mechanisms need to be deeply analyzed thanks to detailed execution traces of experiments.

Case β corresponds to a coverage deficiency of the implemented safety mechanism(s). The safety mechanism is activated, but the propagation of the fault is not mitigated correctly.

Case γ and δ often mean a design or implementation problem since the safety mechanisms have not been activated.

Case γ is simpler to assess. Contrary to Case δ, it corresponds to critical faults, as it leads to the violation of a safety requirement. The non-activation of the safety mechanism can be due to a wrong design, in which this potential cause has been omitted or a wrong safety mechanism has been chosen. It can also be an implementation error: bug in the design of the safety mechanism, a wrong integration.

Case δ, *i.e.*, experiments where no effects are observed, corresponds to several categories that should be analyzed. Firstly, it may result from the injection of "safe fault" (according to ISO 26262, part 5). A **safe fault** is *a fault whose occurrence will not significantly increase the probability of violation of a* ***safety goal***. Hence, the nothing will be observed. Defaults in the fault injection experiments can also be a reason for this last case: *i)* an error remains latent and has not been activated by an experiment scenario, *ii)* the fault has been tolerated by another mechanism or by design (re-initialization of a data corrupted by the injection before using it).

In a conventional fault injection campaign, the output of the experiments is represented as a pie chart composed of the previous categories. Ideally, 100% of errors are detected and recovered, this ensures that the safety requirement is not violated. In reality, some errors are not detected by internal error detection mechanisms, or are not recovered. Upper level safety mechanisms should then prevent the violation of safety requirement.

The observation of the time where the monitored event occurs (*e.g.*, fault injection time, detection, return to a safe state, *etc.*) is important. This observation enables to verify the timing requirements, defined in Figure 1.3 are ensured: Diagnostic Test Interval (most of the time it corresponds to the Detection Time), the Reaction Time and the Tolerance Time Interval (TTI).

To conclude, safety analyses check/recommend safety mechanisms to be put in place, and FI experiments quantify their efficiency, namely detection and recovery coverage. As the safety mechanisms may prevent UEs to be reached, this will enable verifying the safety requirements (and the FFI property) is ensured.

When a safety property is violated one of the two following conclusions holds:

- lack of coverage of a safety mechanism. The implementation of the mechanism should be analyzed in order to improve the coverage, if necessary.
- absence of a safety mechanism leading thus to a revision of the design.

4.2.2 Experiment Traceability

This link between the FIA and the FIE is important in a development process. The definition of the experiments is linked to the assumptions used in the analyses. It is of paramount interest to link the experiments with the safety requirements. The planning of the campaign must include all necessary fault injection experiments to be able at the end to verify that all the requirements have been tested.

4.2.3 Determination of the FIE using FMECA

As we show in the Chapter 3, FIA is linked to FMECA spreadsheets. In this section, the objective is to show how fault injection experiments can be defined from a FMECA row together with S- and Z-shaped causal chains. Then, we define the measures that can be obtained by gathering the experiments from several FMECA rows and how they are selected. This enables obtaining measures of the effectiveness of a safety mechanism or the non-violation of a safety requirement.

4.2.3.1 Definition of Experiments using One Line of FMECA

We consider the FMECA line of Table 2.1 (p.29) has been done at one level L_i. Firstly, the target is an implementation / integration of the specified element at a given architectural level. The analysis of the FMECA starts with the evaluation of the criticality/risk/safety level.

Secondly, the measures have to be assessed. They can be found in the column 5 and 6. Concerning the demonstration that the safety requirements are not violated, we need the description of a physical quantity, variables or signals, involved in the definition of the assertion to ensure that the system is safe. In addition, column 7 helps to identify the safety mechanisms that should be monitored during the experiments. These mechanisms have to be characterized according to their error handling capabilities (detection and recovery).

More generally, if the target implements also upper levels, then following the S-Shaped causal, all safety requirements that may be impacted at each level have to be assessed, that is to say all the safety mechanisms must be evaluated.

The fault model is defined using columns 1, 2 and 3. Firstly, the failure modes of the entities (column 1) have to be considered. Depending on fault injection capabilities on the target, a given

fault/failure mode can be injected easily or not. In the latter case, the Z-shaped causal chain will help determining the potential causes at level L_{i+1} or lower levels, and this way provides means to activate a given fault at multiple levels. This would simplify the implementation of the experiments in many cases, this being an interesting result of the FIA regarding the definition of the FIE.

Concerning the Readouts of the experiments, a similar analysis following the S-shaped causal chain has to be performed. The following columns: failure mode (column 2), local effect (column 4), upper-levels effects with or without safety mechanisms (column 5 and 8), and the safety mechanisms (column 7), must be taken into account in determination of the readouts. First, the failure mode must be monitored to validate that an injected fault is activated. Then, the different effects enable defining the assertions defining the propagation of the fault model, the safe states in order to assess the coverage of the safety requirements. The column 7 focuses on the readouts needed for the assessment of the safety mechanisms.

However, the Activation model can only be defined using the description of the functional requirements, *i.e.* the architectural and behavioral models. The latter gives a detailed specification of the expected activation profile, *i.e.* the software to be developed to perform the experiments, including stubs and drivers for the target component.

4.2.3.2 Example using the Steering Column Locking System

Let's take as an example the line described in Table 4.2 of the Product FMECA of the ESCL.

TABLE 4.2 CONSIDERED LINE OF ESCL PRODUCT FMECA

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety Level	Product Safety Mechanisms (PSM)	Product Effect with PSM
µC-F1: Control the state of the MDB	Erroneous assignment of outputs of the micro-controller ***µC-F1-FM1***	RAM, Flash, ROM Corruption, Oscillator, SW defect...	Spurious activation of the MDB locking state.	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM1:** Watchdog (HW), **PSM2:** 2 different SW modules should be implemented to control the µC-F1 (redundancy)	No effect*

First, we consider that the whole product has been implemented and is now the considered target. There are two important properties to evaluate. First, we must check that no spurious lock is observed since this is a critical Product Undesired Event rated ASIL D. Then, the two safety mechanisms should be assessed to verify that the proposed solution is able to handle the faults leading to the "Failure Modes" identified in the FMECA spreadsheet.

The considered fault model at this level is “erroneous assignment of the outputs of the micro-controller”. The potential causes of this failure mode are RAM, Flash, ROM corruptions from software defects or the µC oscillator defects. A fault injection technique and tools have to be selected to inject faults corresponding to this fault model. In our case, our tool provides appropriate facilities, from SWIFI to Test-port based fault injection techniques (see Chapter 1).

Concerning the Readouts, the target should be monitored to ensure the injected fault/error has been activated/propagated as required for the analysis. Then, different effects and particularly the Upper-level effect have to be monitored in order to verify that a safety requirement is not violated. The non-occurrence of undesired effect should be observed to ensure that the PSMs are efficient. This is the

case of PSM1. In our case study, PSM2 cannot contribute to the readouts of the experiments, as it is only a recommendation for the implementation at a lower level.

To describe completely the experiment, the final step is the identification of the activation model. As already mentioned, the activation model is not integrated in the FMECA spreadsheet but in the description of the functional architecture and the behavior of the product. The behavior of the locking sequence is described in the sequence is described in the sequence diagram of the Figure 4.4.

When the Microcontroller receives a Lock request from the Communication block, then it must follow the following sequence:

- Pre-activation of the MDB: the motor is stopped,
- MDB powers up the motor (Locking): the motor accelerates,
- MDB brakes the motor: the motor decelerates until stopping, and
- MDB returns to an inactive mode: the motor is off.

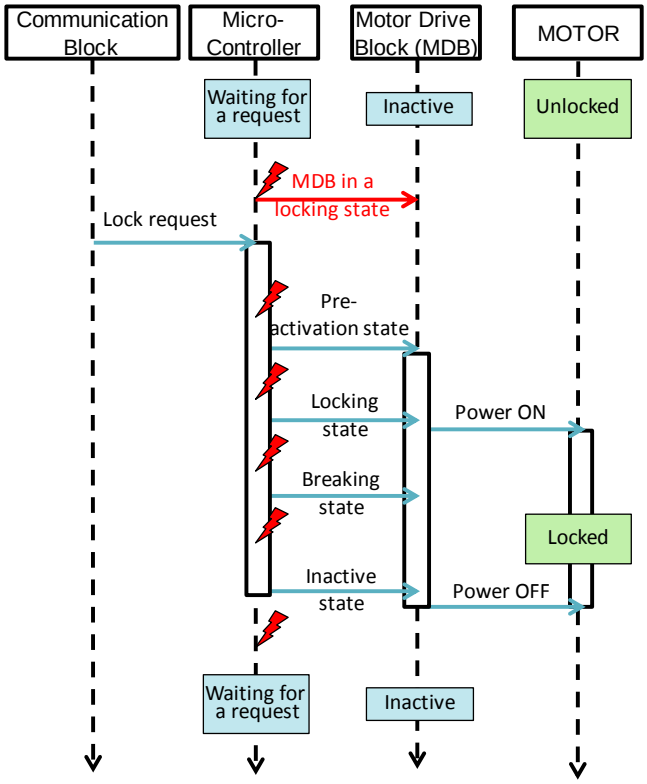


FIGURE 4.4 BEHAVIORAL DESCRIPTION OF THE LOCKING SEQUENCE OF MOTOR AT PRODUCT LEVEL

During this sequence, the motor should reach the Lock position. In the sequence diagram, we identified six states where faults corresponding to the previously defined fault model could be injected during the locking sequence to check the impact on the ESCL product.

4.2.3.3 Definition of Experiments using Multiple Lines of FMECA

We have shown that one line of FMECA helps to determine the experiments that enable assessing the non-violation of a safety requirement and the coverage of safety mechanisms with respect to a fault model. Similarly, the FMECA lines should be gathered in order to globally assess the robustness of

safety mechanisms against the whole fault model of the system (*i.e.*, the set of all the failure modes/potential causes which have been identified), and the non-violation of the safety requirements. The determination of the fault model is of course the main issue. The fault model is the set of potential causes that, through the S-Shaped causal chains, leading to one or a set of UEs.

In the Figure 4.5, we illustrate the assessment of the occurrence of *UE_1* (considered as a critical undesired event). In this case, we can see that two FMECA lines may lead to *UE_1* due to two different entities E_1 and E_2. Then, the causal chain highlights *n* potential causes of these failure modes at the lower lever. Hence, the FIE must gather all the considered failure modes (FM_1, FM_2, FM_3 … FM_n), that may lead to *UE_1*.

The same "causality link" approach can be applied to a safety mechanism, instead of an undesired event, leading thus to the same kind of global analysis of experimental results.

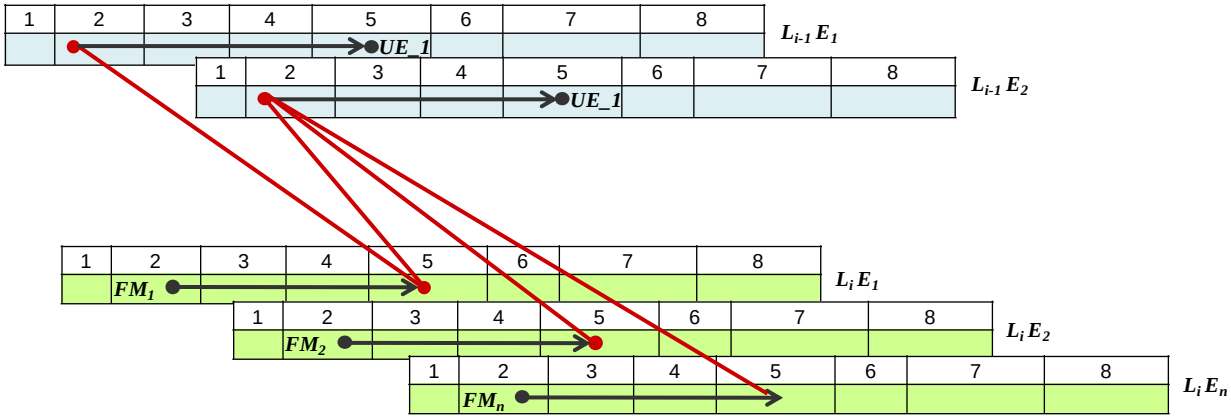


FIGURE 4.5 S-SHAPED CAUSAL CHAIN IN THE DEFINITION OF GLOBAL MEASURES

4.2.3.4 Example using the Steering Column Locking System

In the case of the ESCL, we focus on the evaluation of the non-violation of SG1. Here we consider two levels: system (SCL) and product (ESCL Product). At System level, there are two failure modes that may violate *SG1*: *ESCL_F1_FM1* and *BC_F1_FM2*. Then, considering the ESCL, all the following failure modes will propagate through the S-Shaped Causal Chain: *µC-F1-FM1, CB-F1-FM2, MDB-F1-FM2, MDB-F4-FM1, MDB-F5-FM1, MDB-F5-FM2, SB-F-FM1, SB-F-FM2.*

Finally, in order to verify that safety mechanisms have been well defined to prevent the violation of the safety goal 1, the fault model encompasses all the failure modes identified in the FIA at System and product levels.

We give a summary of this conclusion in the two lines of the System FMECA, hereafter. In

Table 4.3, the two failure modes of ESCL-F1 and BC-F1 are described as well as the potential causes of the ESCL-F1-FM1. Faults leading to all the the failure modes highlighted in red should be injected.

TABLE 4.3 SYSTEM FMECA LEADING TO VIOLATE SAFETY GOAL 1

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety Level	System Safety Mechanisms (SSM)	System Effect with SSM
ESCL-F1: Lock steering column	Spurious Lock **ESCL-F1-FM1**	**µC-F1-FM1** **CB-F1-FM2** **MDB-F1-FM2** **MDB-F4-FM1** **MDB-F5-FM1** **MDB-F5-FM2** **SB-F-FM1** **SB-F-FM2**	Erroneous lock command	Steering column locked while driving **SG1 Violated**	ASIL D	SSM1: Vehicle in motion SSM2: Switched power supply	No effect
BC-F1: Transmit Lock Command from driver's interfaces to ESCL	Unintended BC-F1 **BC-F1-FM2**	Out of our scope	Unintended Lock Command transmits to the ESCL	Steering column locked while driving **SG1 Violated**	ASIL D	SSM1: Vehicle in motion SSM2: Switched power supply SSM4: Plausibility check in the ESCL	No effects

4.2.4 Conclusion on the Identification of the Experiments

At this stage, we have shown how to identify the experiments of the fault injection campaign. These experiments aim at demonstrating the robustness of the safety mechanisms and also at showing the efficiency of the mechanisms to prevent undesired events. We have highlighted the importance of S- and Z-causal chains in the definition of the Readouts and the Measures for the first one and in the definition of the Fault model for the second one. Finally, the Activation model can easily be defined from the behavioral description of the system or a component-system.

4.3 Execution of the Experiments and Evaluation of the Measures

In this section, we first analyze the way FI campaigns are carried out based on the identified experiments. The first objective is the optimization of FIE. Indeed, the final fault model may be very large; further analyses may help reducing the complexity without reducing the validity of the measures. The second objective of this section is the assessment of the fault injection experiments with respect to the safety analyses. A thorough investigation tackles the completeness of the approach.

4.3.1 Optimization of the Experiments

A first objective is the optimization of the number of experiments. There are several dimensions for the optimization. Optimization is frequently tackled in all fault injection studies reported in the literature.

4.3.1.1 Fault Model

Following the conceptual causal chain fault-error-failure, it is worth noting that the faults injected are in practice ***errors***, *i.e.* subtle corruptions of system input and state. Hence, these errors represent a

class of equivalent faults. This is a first way to reduce the number of experiments. This issue has been discussed in (Christmansson & Chillarege, 1996).

When all the faults cannot be exhaustively injected, two strategies can be used. The first one reduces the fault model by selecting specific data type, range of values, boundary values, *etc.* The second strategy is the injection of random faults using a probabilistic approach.

4.3.1.2 Activation Model

The activation model can also be optimized. The main objective is the improvement of the efficiency of the experiment. In most of the cases, a fault could remain non-activated, but also errors may not be propagated. To improve efficiency, the solution is to select the activation profile in order to make sure that the fault will be activated or the error will propagate: considering the corruption of a variable in the memory during the execution of an application, the corruption must be done before the actual reading of the target variable. In this case, the corruption is going to propagate, contrary to cases where the corruption is injected just before writing the variable. In the latter case, the error is overwritten. When these cases are easy to determined, then it is possible to optimize the efficiency of the experiments and to speed up the fault injection experiments. More details are also available in (Christmansson & Chillarege, 1996).

4.3.1.3 Other Testing Activities

In the development process, several testing methods are required. These methods, which can be perceived as fault injection experiments, have two purposes. First, it can be the verification of a safety mechanism. The testing of the functional behaviour of the safety mechanism can be also done by a fault injection approach. Second, robustness tests can be performed and in general lead to generate fault injection tests cases.

This is why fault injection and other testing activities may lead to overlapping tests. An analysis of this issue should avoid the repetition of tests cases and the help optimizing the purpose of each test case. This means that fault injection tests are already carried out in practice, but they are not called fault injection tests. This is something that can be argued to show that the development process of a given provider takes ISO 26262 requirements into account regarding fault injection. But, this remains limited. The work presented in the thesis goes far beyond current tests to comply with ISO 26262 requirements with respect to V&V by fault injection.

4.3.1.4 Results of FIE of Lower Level

Finally, the measures obtained on a target component at lower levels of FIE help reducing of the number of experiments when this target component is integrated into the tested entity. This is similar to unit testing *vs* integration testing. When verifying the integration of the component, it is not necessary to inject the entire fault model defined for this entity. Indeed, the remaining deficiencies of the component should be triggered by fault injection to verify if an upper-level safety mechanism is able to handle these deficiencies. The faults internally mitigated by the component EDC/ERC mechanisms are not interesting at the upper level. This approach is only of interest when the system exhibits different

levels of integration, *e.g.*, when a first target component (a product) is integrated into a system, the measures obtained on the product can be used for the system.

4.3.2 Assessment of the FIA with regards to the FIE

We have shown that the FIA is a guide for the planning of the fault injection experiments. Then, the FIE is used to validate the analyses done in the FIA. What we want to demonstrate in this thesis is the mutual contribution of FIA and FIE.

When the whole FIE campaign has been performed and the measures obtained, the results of the FIE have to be analyzed. At the end of the FIE, there are two types of results: the measures of the error detection and error recovery coverage and the global measures of both *(i)* the coverage of safety requirements (completeness) and *(ii)* the coverage of safety mechanisms (efficiency).

4.3.3 Assessment of one Fault Injection Experiment

We consider an *experiment* in which a set of faults corresponding to the *fault model* is injected, and the *set of safety mechanisms* developed to prevent the propagation of the effects of the fault model. The result of the experiment is either *(i)* the fault is *detected/tolerated (c is the coverage value)*, or *(ii)* a *coverage deficiency* ($\bar{c}$ *is the complement*). Finally, we consider that the *experiment* has been identified in *FIA*.

It should be noted that the non-interference of the fault injection technique or tool in the obtain readouts must be investigated before this assessment.

When at least *one safety mechanism* detects a *fault*, *i.e.*, in the nominal case, the result of the *experiment* is compliant with the *FIA*.

When *experiments* exhibit a *coverage deficiency*, then several causes can be identified:

1. the implementation or integration of a *safety mechanism* is wrong.
2. the implementation of a *safety mechanism* is correct but its definition is incomplete.

In the second case, the solution is the definition of a new appropriate *safety mechanism* to handle the *fault model.*

If a *new safety mechanism* is required, the *FIA* is impacted. Considering a *fault model*, the main question is the following:

Do the observed effects of the *experiment* are the same as those identified in the *FIA*?

If not, it means that the analysis is wrong. The *FIA* should be revised and corrected with the effects observed on the target. Then, a *new safety mechanism* should be identified to prevent the observed effects.

In addition, the new observed effect can lead to identify a new propagation causal chain between levels. For instance, a low level fault may trigger a non-identify failure mode at upper level of abstraction levels. Then, the whole propagation of this new failure mode should be analysed and its criticality determined.

Figure 4.6 summarizes this assessment in a flowchart where the evaluation of the *experiments* leads to modify the implementation of a safety mechanism and to modify the integration of the *safety mechanism,* or need a correction of the *FIA* (*e.g.*, the safety mechanism, the propagation of the fault between levels).

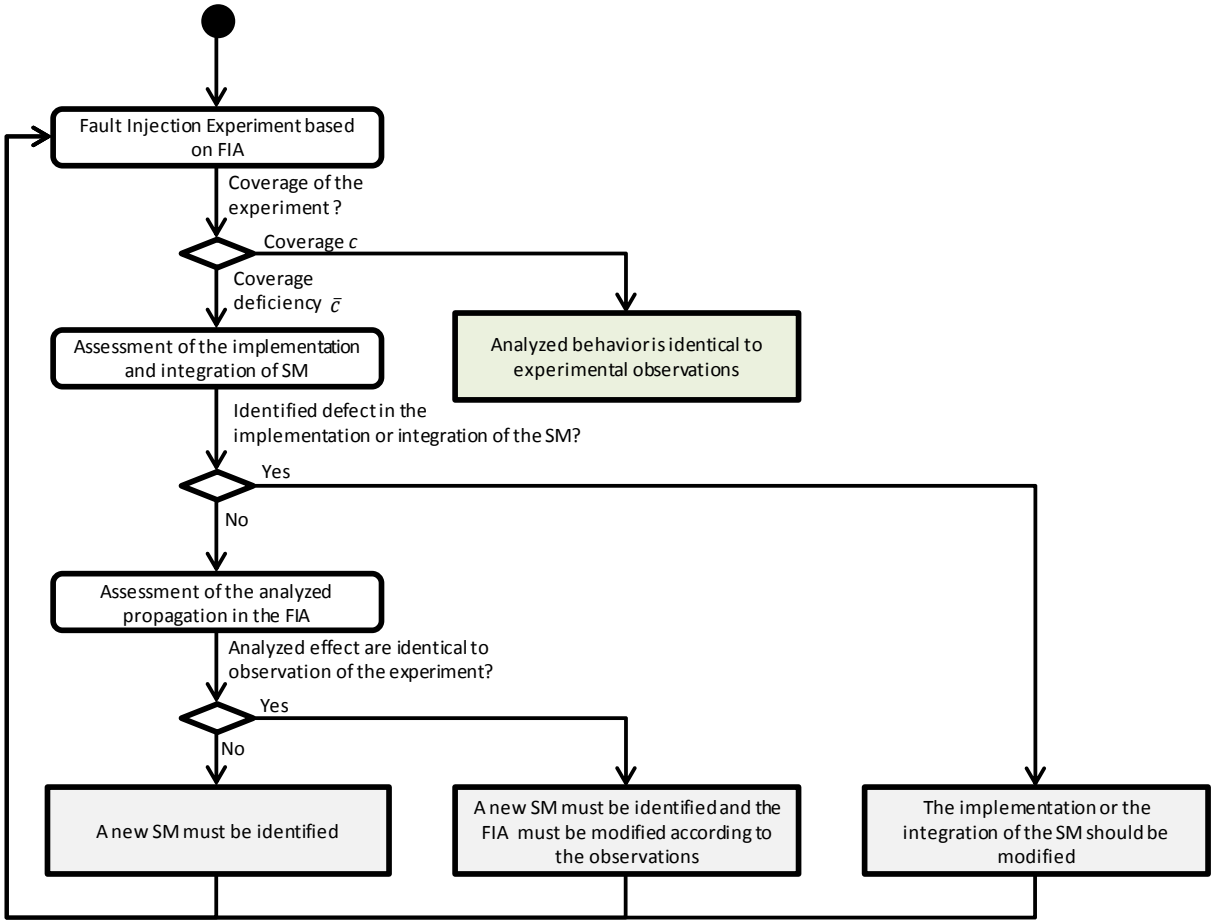


FIGURE 4.6 FLOWCHART OF INTERPRETATION OF FI EXPERIMENTS

4.3.4 Synthesis of the FIE

We have synthesized the FIE flow at product level (as an example) in order to show its interactions with the others activities. Figure 4.7 shows the main steps of FIE for the product level and its interactions with other activities.

The first step of this flow is the definition of the FARM elements based on the results of the Pre-implementation phase: namely the HW and SW Block functional requirements and the results of the FIA. After the definition of the fault injection campaign, the set of experiments can be optimized by reducing their number, thanks to the identification of redundancies with other testing activities, *etc.* At this stage, the fault injection experiments can be run on the target using a fault injection environment. Finally, the assessment of the measures will lead either to the validation of the identified fault tolerance mechanisms or to the modification of the implementation of the target or of the design.

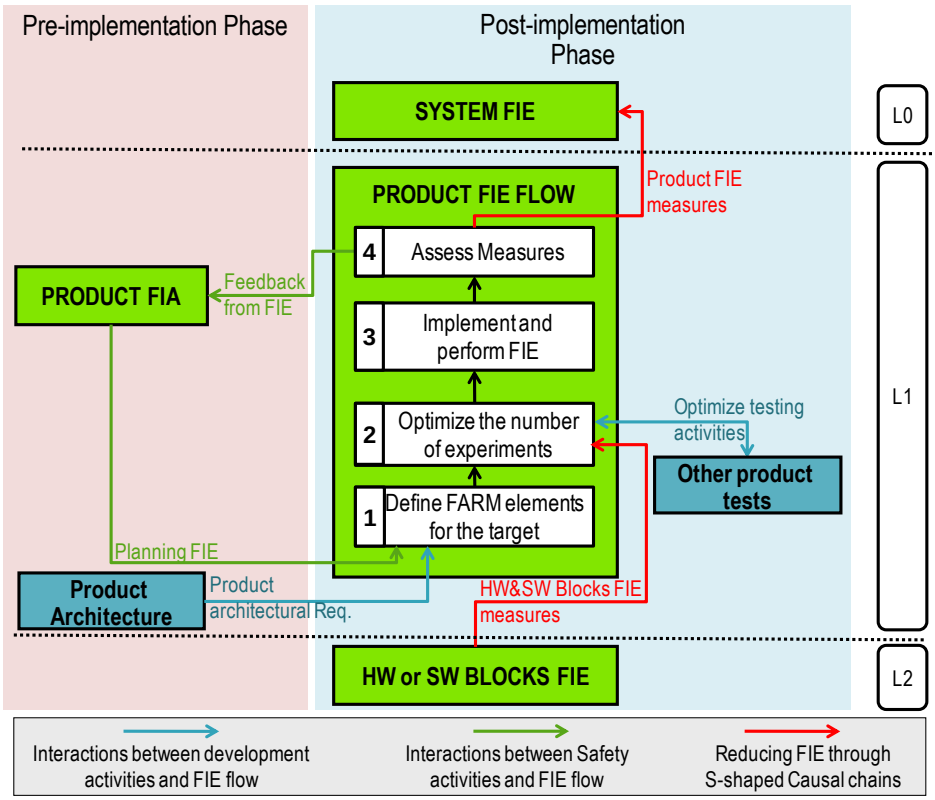


FIGURE 4.7 PRODUCT LEVEL FIE FLOW

4.4 Conclusion

In this chapter, we have demonstrated the usefulness of the FIA in the definition of the FIE. We also tackled issues regarding the contribution of fault injection experiments for the verification and the validation of the safety analyses performed during pre-implementation phase

This chapter raises the issue of the completeness of the experiments definition with this method. In our view, the use of early phase analyses cannot guarantee the completeness of the fault injection experiments. However, a **systematic** approach for definition of the fault injection experiments, based on systematic safety analysis, is at least a concrete guide to the definition of fault injection campaigns. The defined campaigns enable the validation of the proposed safety mechanisms for the prevention of hazards.

The main worry with this method is the definition of an erroneous fault model that does not encompass a complete set of potential causes. However, this problem is not inherent to our approach; all fault injection campaigns may have the same problem. Fault injection campaigns rely on the knowledge of the target or on a specific fault model, and then the main benefit of our approach is to be able to trace the fault model from the beginning of the design down to the experiments. Conversely, FIE results enable to validate, at least partially, the safety analysis done during the pre-implementation phase.

The major difficulty of any fault injection campaign is the definition of the fault model. The injection of large amount of faults may help finding non-analyzed critical paths and/or undesired events. However, this testing approach has often a low efficiency, *i.e.*, most of the tests lead to no observation, and huge campaigns are difficult to analyze. In our approach, we define a set of critical faults. These faults

will more likely produce efficient experiments. The efficiency of the experiments is very important in the industry as a high efficiency reduces testing efforts (time and cost).

The traceability of the experiments is very important in a development process, to make sure that no experiment has been forgotten. In an industrial project, this is a major issue, as complex systems will be developed by several engineers and tested by others. This approach helps the collaboration of safety engineers and test engineers.

Chapter 5 CASE STUDY: FRONT-LIGHT MANAGER

The objective of this chapter is to apply the overall methodology to a representative automotive system. The approach is illustrated on a Front-Light system. This electronic system controls the two headlights of a car. This is a very simple system and this system has only a moderate safety criticality level: ASIL B. At this level, all the requirements on fault injection are not "highly recommended" by the ISO 26262 standard. However, a "proof of concept" of the proposed safety process can be done. In this case study, we focus on the software architectural level and the software module level.

In order to contextualize the SW architecture, a first part is dedicated to the description of the system level and the product level. Fault injection tests are based on assumptions and analyses done at higher levels of architecture. They are determined from the traceability of the requirements leading thus to the identification of efficient fault injection tests cases for the verification and validation of safety mechanisms. The considered targets for the tests are especially the SW architecture and a SW module. A particular attention has been paid to the design of the software architecture. The solution integrates a partitionning between a QM application and an ASIL B application. We are able to tackle a specific problem in the demonstration of safety that is of paramount importance: the verification of the *Freedom From Interferences* (see Chapter 2). This problem is very important because of the integration of multiple software modules with different ASILs on the same platform, a tendency that is growing up in todays complex embedded automotive systems..

5.1 Application of FIA on the Front-Light Manager System

The Front-Light System controls the two Headlights of a car. This is a common automotive case study often used to exemplify AUTOSAR concepts (Fürst, 2008). The proposed design of the Front-Light System is not representative of a real automotive system, as the application is too simple. However, it follows the development process and the design rules of any system according to ISO 26262.

The architecture of the Front-Light System is depicted in Figure 5.1. A 12V battery powers the Front-Light System. Its main function consists in the control of the two headlights of the car to light the road at night, in tunnels. It also controls an indicator on the Dashboard. This indicator signals to the driver the state of the headlights. The Front-Light System communicates with the Dashboard ECU through the CAN network of the car.

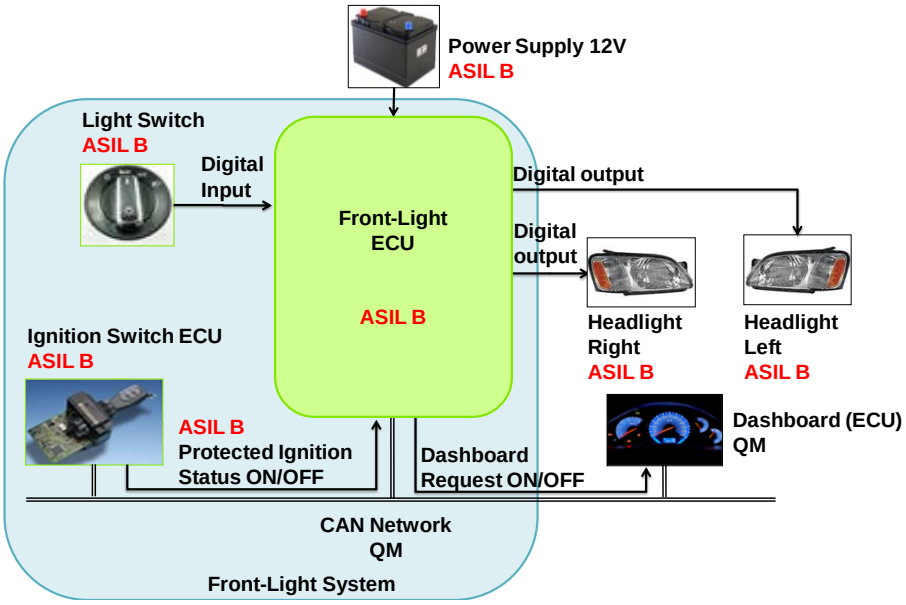


FIGURE 5.1 ARCHITECTURE OF THE FRONT-LIGHT SYSTEM

A Preliminary Hazard Analysis (PHA) identifies one Undesired Event: UE01, "***Loss of the Headlights***", rated ASIL B

The ASIL is allocated by estimating the severity, exposure, controllability (Table 5.1) of the Undesirable Event, using the ASIL Matrix in Chapter 1.

TABLE 5.1 FRONT-LIGHT SYSTEM'S UES ASIL ALLOCATION

UEs	Situation	Severity	Exposure	Controllability
UE01 Loss of the Headlights	Night/tunnel. No street lights	S3: Life-threatening injuries (survival uncertain), fatal injuries	E2: Low probability: at night/tunnel without street lights	C3: Difficult to control or uncontrollable

The PHA defines the following ASIL B safety goal.

- ***SG1: The system shall not spuriously cut off both Headlights (ASIL B)***

The following section will describe the FIA process at System, Product and Software levels. The process will highlight how the system should be designed to ensure the two safety goals, and will prepare the fault injection experiments definition.

5.2 FIA at System Level: Front-Light System

The Front-Light System, depicted in Figure 5.1, encompasses four products. The main product is the Front-Light ECU, which gathers information from Ignition Switch ECU and Light Switch ECU in order to set the Headlights On or OFF and to light the Dashboard indicator. The control logic of the Front-Light ECU is very simple: it must light the two headlights of the car and the Dashboard indicator when both Light Switch status and Ignition switch status are ON.

All the product functional requirements are synthesized in Table 5.2

TABLE 5.2 DESCRIPTION OF THE FUNCTIONS OF THE FRONT LIGHT SYSTEM.

Product	Product Function Id #	Product Functional Requirements
Front-Light ECU	FL-ECU_F01	Front-Light ECU must send Dashboard State (ON/OFF) through the CAN Network
	FL-ECU_F02	Front-Light ECU drives the Headlights state (ON/OFF) in less than 600ms
Light Switch	LS	The Light Switch provides a ON/OFF signal to the Front-Light ECU
Ignition Switch ECU	IS-ECU	The Ignition Switch ECU must send periodically the Ignition Switch Status (ON/OFF) through the CAN Network to the Front-Light ECU
CAN Network	CAN-F01	Transmit Ignition Switch Status from Ignition Switch ECU to Front-Light ECU
	CAN-F02	Transmit Dashboard indicator status from Front-Light ECU to Dashboard ECU

The FMECA spreadsheet, given in Table 5.3, summarizes the fault injection analysis carried out for the six functional requirements.

TABLE 5.3 FMECA OF THE FRONT-LIGHT SYSTEM

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety	Element	Failure Modes
FL-ECU_F01	Loss of Dashboard indicator Status		No Dashboard Indicator Status sent	The Dashboard Indicator Status is not lighted according to the specification	QM		
	Erroneous Dashboard indicator Status sent		Erroneous Dashboard Indicator Status sent	The Dashboard Indicator Status is not lighted according to the specification	QM		
FL-ECU_F02	Unintended Headlight state ON		Unintended Headlight state ON	Discharge of the battery	QM		
	Loss of Headlights state		Loss of Headlights state	Loss of the Headlights	ASIL B	Fail-safe implementation of the FL ECU	ASIL B
LS	Loss of Light Switch signal		Light Switch Status ON not sent to the FL-ECU	Loss of the Headlights	ASIL B	Check done by the FL-ECU (SW mechanism)	No effect *
	Erroneous value ON sent		Light Switch Status OFF not sent to the FL-ECU	Discharge of the battery	QM		
IS-ECU	Loss of Ignition Switch Status		Ignition Switch Status ON not sent to the FL-ECU	Loss of the Headlights	ASIL B	Check done by the FL-ECU (SW mechanism)	No effect *
	Unintended sending of the Ignition Switch Status ON		Erroneous Ignition Switch Status ON sent to the FL-ECU	Discharge of the battery	QM		
	Erroneous Ignition Status frequency higher than expected (timing error)		CAN saturation	Loss of the Headlights	ASIL B	FL-ECU should diagnostic the CAN Network for Ignition Switch Status	No effect *
CAN-F01	Loss of CAN communication		Ignition Switch Status ON not received by the FL-ECU	Loss of the Headlights	ASIL B	CAN data integrity (combination of CRC, Frame counter, timeout)	No effect *
	Erroneous CAN communication (interferences)		Corrupted Ignition Switch Status received by the FL-ECU	Loss of the Headlights	ASIL B	CAN data integrity (combination of CRC, Frame counter, timeout)	No effect *
CAN-F02	Loss of CAN communication		Dashboard indicator status ON not received by the FL-ECU	The Dashboard Indicator Status is not lighted according to the specification	QM		
	Erroneous CAN communication (interferences)		Erroneous Dashboard Indicator Status received by the DB-ECU	The Dashboard Indicator Status is not lighted according to the specification	QM		

* assuming a perfect coverage of safety mechanisms

In the FMECA spreadsheet, we can identify six failures modes that may lead to violate the safety goals. Most of the proposed mechanisms cannot be completely determined at this level. These mechanisms will be refined at underlying levels. For example, the failure modes of the FL-ECU will not be handled at System level but the choice has been made to design a fail-safe FL-ECU in order to mitigate the occurrence of failure modes.

In the spreadsheet given above, we have also identified that a checking of the outputs of LS, CAN and IS-ECU must be verified to ensure that they are valid.

We also see that specific analyses must be done on the CAN network. Indeed, it transmits safety critical information that may lead to violate a safety goal. It should be designed according to ASIL B. However, we consider the bus CAN to be QM. Then, an ***End to End—E2E*** protection is needed to keep safe the critical signal from IS-ECU to FL-ECU. The purpose of E2E protection is to prevent the data through serial communication from corruption, deletion, repetition, insertion, incorrect sequence, delay, masquerading. The E2E protection involves CRC, time out monitoring or counter. The implementation of the E2E protection is sufficient to fulfill the safety requirements (ASIL B) of the communication according to the ISO 26262.

5.3 FIA at Product Level: Front-Light-ECU

We focus on the development of the Front-Light ECU product. By analyzing the results of the FMECA, we identify one line of the Front-Light ECU that lead to the violation of the safety goal. This failure mode will later be referred to as Product-undesired event, "***P-UE01: Loss of Headlights state***"

At this, level,we consider the micro-controller that runs the software applications on the Front-Light ECU. Because the inputs and outputs at product level are the same as those at system level, the failures at product level are those observed at system level. Looking more carefully to the failure at product level, in this simple example, the failure mode at product level are directly due to onefailure mode at SW block level. P-UE01 can be considered as a SW-UE, software undesired event (SW-UE01).

The architecture of the Product level is described in Figure 5.2. The Front-Light Software Architecture implements two functions, *(i)* the control of the Headlights status (FL-ECU_F02)—ASIL B, *(ii)* the control of the Dashboard Indicator Request (FL-ECU_F01)—QM. The Software Architecture must also manage the inputs from the Light Switch (ASIL B) and from the CAN network (QM). Hence the Front-Light Software Architecture integrates mixed ASILs modules.

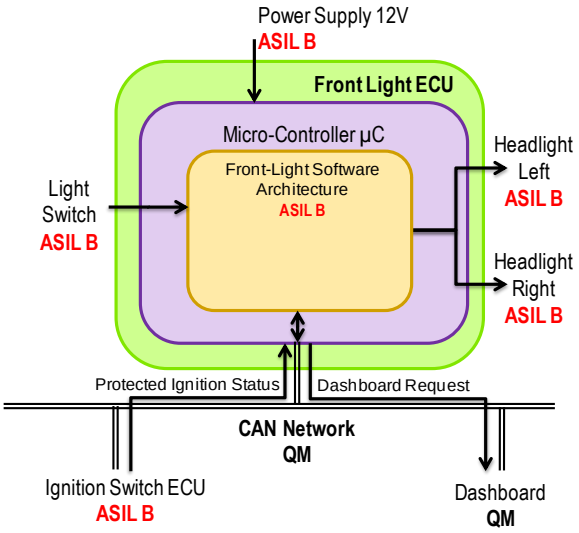


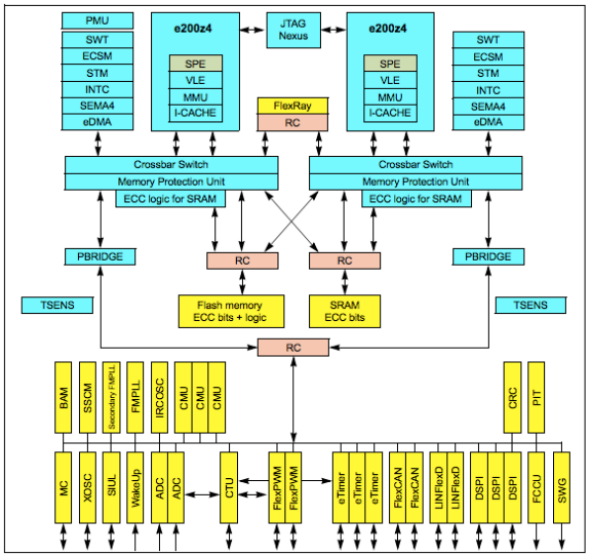
FIGURE 5.2 ARCHITECTURE OF THE FRONT-LIGHT ECU

Two interesting issues must be investigated at this level. These two issues are raised by the ***implementations dependencies*** of the hardware blocks and the software blocks. First, as we will not

investigate in more details the FIA of the hardware, we will at least evaluate the chosen micro-controller. Then, we can easily identify that the Software architecture must integrate two functions with differents ASIL levels. Hence, two solutions are proposed following the ISO 26262 standard: *i)* the development of the two applications according to the highest level of criticality, or *ii)* the integration of the two applications with different ASILs and a demonstration of the Freedom From Interference.

5.3.1 Safety Analysis of the Micro-Controller

The chosen micro-controller is a Leopard SPC56EL70 (STMicroelectronic, 2013). This model is based on a PowerPC architecture and includes two identical cores (e200z4d cores) connected to a single shared main memory. This architecture as been defined by the manufacturer to design a µC that fullfil the architectural metrics and the PMHF for ASIL D. The architecture is illustrated in Figure 5.3. The microcontroller can be configured into lockstep or decoupled modes.



ADC – Analog-to-Digital Converter
BAM – Boot Assist Module
CMU – Clock Monitoring Unit
CRC – Cyclic Redundancy Check unit
CTU – Cross Triggering Unit
DSPI – Serial Peripherals Interface
ECC – Error Correction Code
ECSM – Error Correction Status Module
eDMA – Enhanced Direct Memory Access controller
FCCU – Fault Collection and Control Unit
FlexCAN – Controller Area Network controller
FMPLL – Frequency Modulated Phase Locked Loop
INTC – Interrupt Controller
IRCOSC – Internal RC Oscillator
JTAG – Joint Test Action Group interface
LINFlexD – LIN controller with DMA support
MC – Mode Entry, Clock, Reset, & Power
PBRIDGE – Peripheral bridge
PIT – Periodic Interrupt Timer
PMU – Power Management Unit
RC – Redundancy Checker
RTC – Real Time Clock
SEMA4 – Semaphore Unit
SIUL – System Integration Unit Lite
SSCM – System Status and Configuration Module
STM – System Timer Module
SWG – Sine Wave Generator
SWT – Software Watchdog Timer
TSENS – Temperature Sensor
XOSC – Crystal Oscillator

FIGURE 5.3 SPC56EL70 ARCHITECTURE (STMICROELECTRONIC, 2013)

In lockstep mode, the two cores run the same instructions and the Redundancy Checker (RC) compares the results. It is required to ensure the safety of critical functions (*e.g.*, ASIL C and D) by offering a tolerance to transient hardware faults. It can be noted that the safety level has been demonstrated by the manufacturer as a Safety-Element out of Context (SEooC). The decoupled mode enables different instructions to be executed on each core, and therefore execute several tasks in parallel. This mode can be used in order to enhance the performance of the application and/or implement safety mechanisms.

The lockstep mode has been adopted. This configuration of the µC enables handling most of the CPU errors due to single Event Upsets (SEU) in the CPU.

5.3.2 Freedom From Interferences Analysis

The FFIA (Freedom From Interferences Analysis) enables several causes of malfunction of the higher ASIL functions to be identified, by analyzing the interference propagation channels. These

interferences are by definition caused by the lower ASIL functionalities (here, QM funtions). The identified causes also require mitigation means –definition of safety mechanisms– in order to prevent the violation of safety goals.

Without going into details, two applications with different criticality levels have been allocated on the microcontroller. There are an ASIL B application (the Headlight command and the Light Switch input signal management) and a QM application (Dashboard Indicator Signal and CAN management).

In practice, the QM application must not interfere with the ASIL B application following these channels:

1. **Real-time behavior Interferences**: *e.g.*, erroneous execution of the QM application (excessive execution time, erroneous period)
2. **Service Calls Interferences**: *e.g.*, wrong input provided by the QM application to ASIL B application.
3. **Shared Data Interferences**: *e.g.*, corruption by the QM application of a critical data used by the ASIL B application.
4. **Shared Memory Interferences**: *e.g.*, corruption of Critical data by the QM application through shared-memory (ROM, RAM, stack)

If the QM application interferes with the critical application a safety requirement may be violated. Hence, the safety application should be protected up to an ASIL B against these interferences.

To enable the correct execution of the critical part of the software, temporal and spatial partionning must be implemented. These mechanisms must protect the computational and communication channels from the interferences due to non-safety software.

- ***Spatial Partitioning*** ensures that one software module cannot alter the code or private data of another software module. It also prevents a software module from interfering with the control of external devices (e. g., actuators) of other software module.
- ***Temporal Partitioning*** ensures that a software module cannot affect from a timing viewpoint the ability of other software modules to access shared resources, such as the network or a shared peripheral. This includes the temporal behavior of the services handling such resources (latency, jitter, duration of resource usage during an access).

These two kinds of mechanisms must be implemented. However, the solutions retained cannot be described precisely at product level as they depend on the detailed software architecture.

It is important to understand that the interferences given above are due to the implementation of the application requirements. They introduce dependencies between applications that can lead to failures. Such failure modes cannot be analyzed in the FIA since the FIA is carried out at a more abstract level.

Each interference model can be considered as a failure mode of the partitioning mechanism in presence of a potentially corrupted QM application (worst case). These failure modes may all lead to violate the P-UEs. They are respectively referenced to as: SWB-UE02 to SWB-UE05. These software blocks undesired events are all rated ASIL B as their occurrences can cause the violation of SG1 or

SG2. All the SWB-UEs are reported in Table 5.4. These failure modes will have to be refined at software block architecture.

TABLE 5.4 SOFTWARE BLOCK UNDESIRED EVENTS

Software-UEs #	Failure Mode Description	ASIL
SWB-UE01	Loss of Headlights state	ASIL B
SWB-UE02	Real-time behavior Interferences	ASIL B
SWB-UE03	Service Calls Interferences	ASIL B
SWB-UE04	Shared Data Interferences	ASIL B
SWB-UE05	Service Calls Interferences	ASIL B

5.4 FIA at SW Block Architectural Level

5.4.1 AUTomotive Open System Architecture – AUTOSAR

AUTOSAR (AUTOSAR, 2015) is a standard for automotive E/E software architectures developed by major OEMs and suppliers. The *core partners*, which pilot the consortium, include Bosch, Continental, BMW, Volkswagen, PSA, Ford, General Motor, Toyota and Daimler Chrysler and. hundred partners called *premiums members*, including Valeo, participate to the drafting of the specification of the software modules. The *associated members* can use the standard. Today, AUTOSAR is a major trend of software development in the automotive industry.

AUTOSAR supports an application-specific approach for automotive software development as opposed to an ECU-specific one. This approach provides means for developing applications that are platform independent as long as they abide by a specified process and the interfaces provided. The AUTOSAR architecture mainly encompasses an application layer (comprising Software Components (SW-C), a Run-Time Environment (RTE) and the Basic Software (BSW).

The BSW is composed of three main layers: Service Layer, ECU Abstraction Layer, and Microcontroller Layer (Figure 5.4). These layers are decomposed into five stacks (each stack is cross-layer):

- the Service stack,
- the Memory stack,

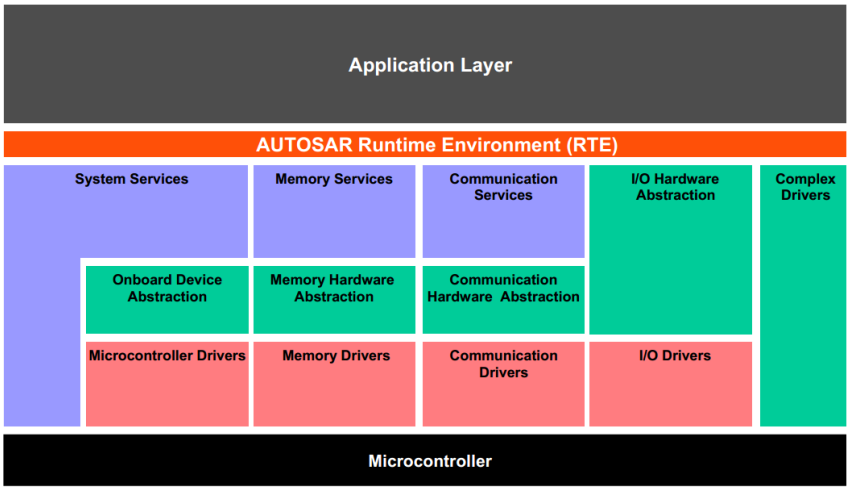


FIGURE 5.4 DESCRIPTION OF AUTOSAR LAYERS AND STACKS OF THE BASIC SOFTWARE (AUTOSAR, 2015)

- the Communication stack,
- the Input/Output Hardware Abstraction stack,
- and the Complex Devices Drivers stack.

One important component of the BSW is a priority-based task scheduler (called AUTOSAR-OS), each task being composed of application runnables belonging to SW-Cs. A runnable is a processing step belonging to a SW-C (a C function) that can be periodic and/or aperiodic. Runnables can be connected through the RTE for data communication. Runnables are mapped to tasks depending on their characteristics (*e.g.* period, input data scheme, *etc.*). In practice, AUTOSAR-OS is a module of the basic software, derived from the OSEK/VDX Kernel, enabling the scheduling of tasks and Interrupt Service Routines—ISRs.

5.4.2 Partitioning Concept in AUTOSAR

In order to implement the partitioning requirements imposed by FFIA, the following AUTOSAR concepts and modules shall be involved.

OS-Application: The AUTOSAR-OS offers the possibility to group different OS objects (Tasks, ISRs, Alarms, *etc.*) into so called OS-Applications. All objects within one OS-Application share their memory protection scheme and the access rights.

According to AUTOSAR-OS Specifications (AUTOSAR_SWS_OS, 2014), OS-Applications can either be trusted or non-trusted. Trusted OS-Applications are allowed to run in CPU Supervisor Mode without restrictions and non-trusted ones are running in CPU User Mode with limited access to OS and HW resources.

It should be noted that “trusted” and “non-trusted” definitions do not match with “safety” and “non safety”. In a simple case, *i.e.* when there are only one safety OS-application and one QM OS-application implemented, then the “trusted” application is the safety one. The QM one is “non-trusted” and requires to be runned with limited access.

In other cases, it can exist multiple OS-Applications with different ASILs. Then, the “non-trusted” mode shall be divided into multiple instance to protect separately each OS-Application.

The OS-Application enables both spatial and temporal partionning to be implemented, at least partially.

MMU/MPU: The basic memory protection requirement to be fulfilled by the OS is to segregate data, code and stack sections of an OS-Application. In the AUTOSAR OS standard, this protection is activated during the execution of the non-trusted OS-Applications in order to prevent the corruption of the trusted OS-Application memory sections. Moreover, the MMU/MPU can also be used to protect private data and stack within the same OS-Application if necessary.

The memory protection relies on a hardware support (MMU/MPU) integrated in the microcontroller. The MPU/MMU provides spatial protection of the memory.

AUTOSAR Inter OS-Application Communicator – IOC: The communication between two OS-Applications has also to be protected. Indeed, OS-Applications intend to create memory protection boundaries, therefore dedicated communication mechanisms are needed to cross them. This feature is implemented in AUTOSAR-OS and called IOC (AUTOSAR_SWS_OS, 2014). It is the dedicated

communication mean between OS-Applications, whether or not the OS-Applications are allocated to the same core (the communication can be between two OS-Applications on the same core, or allocated to two different cores in multi-core architectures). Its main function is to ensure the integrity of the transmitted messages via a buffer. These messages can be data structures or notifications (activation of a task, callback…).

5.4.3 Software Architecture of the Front-Light Manager

The software architecture is illustrated in Figure 5.5. The execution of the software is controlled by AUTOSAR-OS that needs to be developed according the highest ASIL of applications running on the microcontroller, *i.e.* ASIL B in our example.

According to the requirements on spatial and temporal partitioning to ensure the FFI, two OS-Applications are defined. The first one manages the safety critical functionalities (ASIL B) and the second one, non-safety (QM) functionalities. These OS-Application will be respectively referred as: `Front-Light OS-Application` and `DemoApp OS-Application`. This implies that all the modules of the critical OS-Application should be developped according to ASIL B.

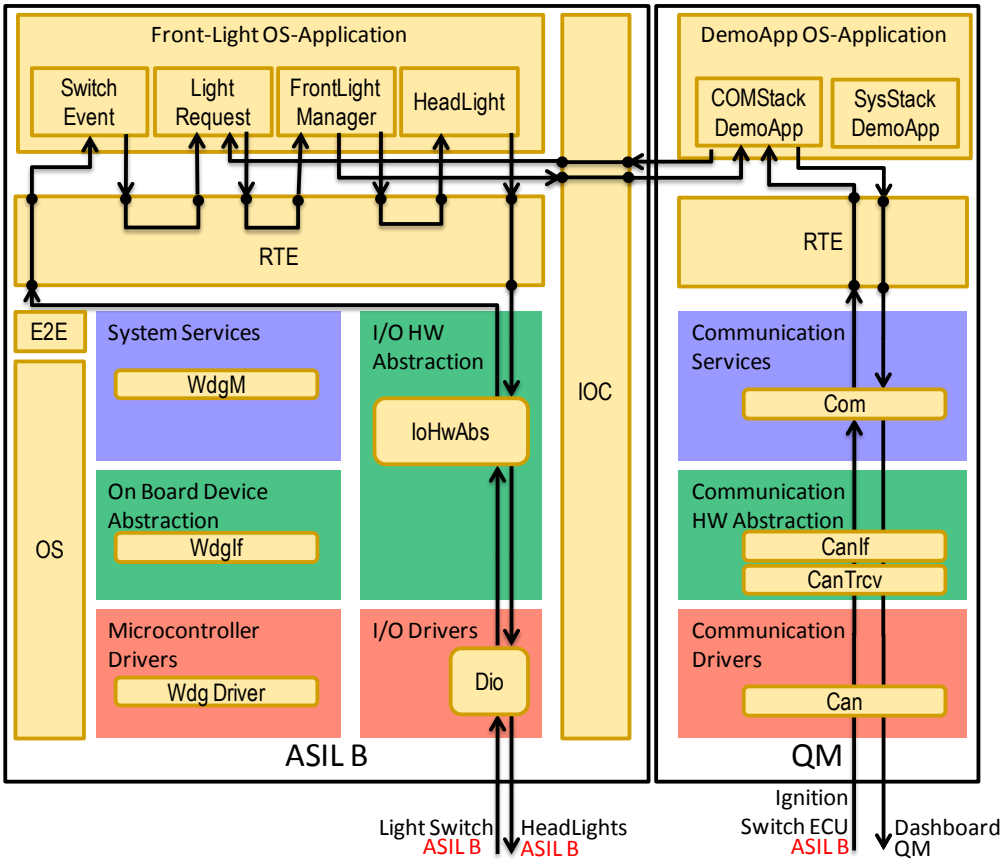


FIGURE 5.5 FRONT-LIGHT SOFTWARE ARCHITECTURE

The `Front-Light OS-Application` ASIL B is composed of four Software Components (SW-C) (*Switch Event*, *Light Request*, *Front-Light Manager*, and *Headlight*). These modules aim at producing both headlights and dashboard indicator output according to the inputs received from the Ignition Switch and the Light Switch. Their behavior will be detailed in the next Section.

The basic software - BSW (Systems Services, IO HW Abstraction Stack, *etc.*) is ASIL B and enables reading the input information from the Light Switch, and it enables the outputs of the µC to be set for the Headlights.

An IOC is used by the `DemoApp OS-Application` to communicate with the `Front-Light OS-Application`. Inputs from the QM OS-Application received by the ASIL B OS-Application must be checked. The E2E protection mechanisms is used to protect the ignition switch status. Here, it enables the data received through the CAN to be unwrapped and validated (this requirement comes from the System FIA).

Besides, the `DemoApp OS-Application` QM is composed of two SW-Cs (*ComStackDemoApp* and the *SysStackDemoApp*): *(i) ComStackDemoApp* is involved in the dispatching of messages between CAN and SW-Cs, *(ii) SysStackDemoApp* is an independant SW-C in the functional behavior of the application that is representative of others possible SW-Cs integrated into the software architecture.

In both cases, the RTE plays its role of communication middleware between software entities SW-C. In both cases, it is composed of pre-defined communication channels interconnecting software components belonging to the same OS-Application, either the `Front-Light OS-Application` or the `DemoApp OS-Application`. It is important to mention that each OS Application has its own instance of RTE managing the interaction between the SW-Cs previously described. The *Communication Stack* that handles CAN transmissions and receptions is located within the QM OS-Application.

Finally, the micro-controller enables addressing space protection through the MPU; hence, the execution of the `DemoApp OS-Application` is run in protected mode to prevent incorrect access to the Front-Light OS-Application memory space.

5.4.4 Behavioral Description of the Application

The objective of this section is to describe the behavior of the Application. The Software architecture is based on the AUTOSAR OS.

The **safety critical OS-Application**, *i.e.* the `Front-Light OS-Application`, is composed of two tasks.

Task 1 encompasses the runnables of the safety critical software modules. The runnables are executed in the following order every 10 ms:

- ***Switch Event:***
  - *CheckSwitch():* It reads and checks the value of the Light Switch from the IOHWAbs through the RTE, and sends a checked status to the Light request module through the RTE.
- ***Light Request:***
  - *Check_Plausibility():* It unwraps the E2E protection, reads and checks the value of the Ignition Switch from the ComStackDemoApp through the IOC and then compares this value with the value received from the Switch event module. Finally, it writes the result in the global variable *u8PlausResult*.

- *Request_Light():* Based on the value of *u8PlausResult*, this runnable sends the command to the Front Light Manager through the RTE.

- ***Front-Light Manager**:*
  - *Request_Check():* It reads and checks the command from the Light request. Then, it sends the command for the indicator to the COMStacksemoApp through the IOC, and also sets a global variable *u8IsReqValid* with the command for the headlight.
  - *Set Light():* It reads the value of *u8IsReqValid* and sends the command to the Headlight module through the RTE.
- ***Headlight*:**
  - *Set Command():* It reads the value sent by the Front-Light Manager module and then sends the Command to the IO HW Abstraction module through the RTE.

- ***IO Hardware Abstraction***
  - *IOHWAbs_ReadWriteUpdate():* The module switches the headlights throught the DIO (*Digital Input Output*) channels based on the command from Headlight module. It also reads the inputs of the Light Switch from the DIO and sends the value to the Light Switch module through the RTE.

Task 2 contains the main functions of the basic software except COM stack. Each function is called according to a specific periodic timing event at 10 ms (Watchdog Mgr, ECU Mgr, BSW Mgr, diagnostic event managers, development error tracer, *etc.*). This task provides low level services for the execution of our critical OS application.

The **QM OS-Application** is composed of one task which encompasses the COMStack_DemoApp and the COM. This task is periodic at 5 ms.

The **COM stack** manages the communication between the Front-Light ECU and the other ECUs through the CAN network. It receives and sends the messages defined by the COMStack_DemoApp.

The **COMStack_DemoApp** transmits the data between the SW modules of the safety critical OS-Application and the COM Stack. It transmits the ignition switch wrapped signal to the Light Request module and it transmits the Dashboard indicator status to the COM stack.

5.4.5 FIA of the Software Architecture

A FMECA has been done to detail the fault propagation paths through the software architecture.

We have considered 8 software modules that perform 26 functions. To simplify the analysis, the RTEs and the IOC failures have not been considered in our analysis but their failures may affect the RTE and the wrapped function. We have also decided to focus on one SW module only: the Front-Light Manager Module that is highly critical in the `Front-Light OS-Application`.

It is worth noting that the complete table is composed of 116 lines. An extract of the complete spreadsheet, which focus on the Front-Light Manager Module, is given in Table 5.5. The failure modes considered are timing errors (*e.g.*, task period too fast or too slow, an erroneous scheduling, an execution timeout), data errors (Corrupted data like out of range, valid error, or data loss), function call errors (function not called, function call with wrong arguments).

TABLE 5.5 SOFTWARE FMECA OF THE FRONT-LIGHT MANAGER MODULE (SUBSET OF THE FMECA)

Element	Failure Modes	Potential Causes	Local effects	Upper-level effect	Safety level	Software Safety Mechanisms (SM)	Upper-level effect with SM
Front-LightManager must (periodically 10 ms) read the provided light request	Period too slow		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM alive monitoring 2 (10 ms)	Reset + Safe Mode
	Erroneous Scheduling (Before/After)		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM Control Flow	Reset + Safe Mode
	Execution Timeout (more than designed)		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM Deadline Monitoring	Reset + Safe Mode
	Erroneous data red		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	Range checks of input and output data	Lack for Valid Errors ASIL B
Front-LightManager must (periodically 10 ms) refresh u8IsReValid	Period too slow		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM alive monitoring 2 (10 ms)	Reset + Safe Mode
	Erroneous Scheduling (Before/After)		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM Control Flow	Reset + Safe Mode
	Execution Timeout (more than designed)		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM Deadline Monitoring	Reset + Safe Mode
	data not refreshed		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	WdgM alive monitoring 2 (10 ms)	Lack for Valid Errors ASIL B
	Erroneous data refresh		Erroneous u8IsReqValid used	SWB-UE01	ASIL B	Range checks of input and output data,	Lack for Valid Errors ASIL B
Front-LightManager must (periodically 10 ms) send Dashboard Request to the COMStackDemoApp throught IOC based on u8IsReqValid	Period too slow		Erroneous Dashboard Request used	SWB-UE03 SWB-UE04	QM		
	Erroneous Scheduling (Before/After)		Erroneous Dashboard Request used	SWB-UE03 SWB-UE04	QM		
	Execution Timeout (more than designed)		Erroneous Headlight command used	SWB-UE01	ASIL B	WdgM Deadline Monitoring	Reset + Safe Mode
	No data sent		Erroneous Dashboard Request used	SWB-UE03 SWB-UE04	QM		
	Erroneous Request sent		Erroneous Dashboard Request used	SWB-UE03 SWB-UE04	QM		

These failures may have multiple causes:

1. Software (systematic) faults:
    a. wrong design of a software module
    b. wrong design of the software architecture
    c. wrong implementation of the requirements (including interferences)
2. Hardware failures: Corruption of the µC memories (RAM, ROM, registers, not handled by Error Correcting Codes).

Safety mechanisms have been identified in order to handle the failure modes of the software modules. Three alive supervision functions of the WdgM are configured to check that the critical SW-Cs are still executed. Alive supervisions of ComStackDemoApp and theCan stack are implemented to prevent interference on the Ignition Switch status provided to the Light Request. Control flow and deadline supervision functions are implemented to monitor the execution of the ASIL B SW-Cs.

It should be noted that propagation of the above mentioned failures are risky in two cases:

- ***Use case 1***: the headlights are already ON and the user does not change the inputs (Light Switch OFF or Ignition OFF). In this case the loss of the headlights violates the safety goal.
- ***Use case 2***: the headlights are OFF and the user wants to change the state to ON. In this case the safety goal may be violated if the lights are not put ON when requested by user inputs (Light Switch ON or Ignition ON). In this case the loss of the headlights violates the safety goal.

It is assumed that the response time to light the headlights must be less than 600 ms. The application must reach the intended state (headlights ON in the considered use cases) within this time window.

Finally, we can observe in the FMECA that some failure modes have not been completely handled. In this application, valid errors are potentially provided by several modules in the critical path. These valid errors correspond to wrong values transmitted by a module, but the wrong value is within a valid range. Then, the error cannot be detected by our architecture. They may lead to a safety requirement violation. This lack of coverage has been neglected in this application example because of its low probability of occurrence. This situation may be handled by a slight re-design and the inclusion of fine-grain data checks and/or more complex runtime assertions.

5.5 S-Shaped Causal Chain

In this section, our objective is to illustrate how fault injection can be planned following the S-shaped Causal chain in this Front-Light Manager application

First, we isolated the first line of Table 5.5 and we traced the propagation of the cause through product and system level in Table 5.6.

The S-shaped causal chain highlights that the considered failure mode “period to slow of the Front-Light Manager” may lead to the violation of the safety goals SG1. This critical path is highlighted red in Figure 5.6. There are two identified mechanisms to recommend from this threat: the use of a robust microcontroller and the integration of alive supervision in the WdgM software module.

The tables given above are extracted from the complete FIA analysis. In the low level reported in the tables, we can see that a safety mechanism has been identified: the WdgM. In reality, several safety mechanisms have been identified: alive monitoring, deadline monitoring and control flow checking. All these individual safety mechanisms are implemented with the WdgM, a generic module providing such safety mechanisms and that can be configured for a given application (window period the entity is alive, deadline values, reference control flow graph). In the last chapter of the thesis, the WdgM will be the target for the experiments.

Hence, to demonstrate the coverage of the safety requirement, the injection of potential causes of the failure mode should demonstrate that there is no impact on the critical application outputs and that the WdgM detects it and handles it correctly.

As soon as the fault injection target has been identified, experiments must be defined following the FARM model explained in Chapter 1. Following this FARM model, we define here the experiments that must be carried out.

TABLE 5.6 ILLUSTRATION OF THE S-SHAPED CAUSAL CHAIN

System Element	Failure Modes	Potential Causes	Local effects	Upper-level effect	Safety level	System Safety Mechanisms (SSM)	Upper-level effect with SSM
FL-ECU_F02	Loss of Headlights state ***FL-ECU_F02_FM1***	***FL-SW_F02_FM1***	Loss of Headlights state	Loss of the Headlights ***SG1***	ASIL B	Fail-safe implementation of the FL ECU	ASIL B

Product Element	Failure Modes	Potential Causes	Local effects	Upper-level effect	Safety level	Product Safety Mechanisms (PSM)	Upper-level effect with PSM
The Software architecture must handle the Headlights status ***FL-SW_F02***	Loss of Headlights status ***FL-SW_F02_FM1***	***FLM_F01_FM1***	Loss of Headlights state	Loss of the Headlights FL ***FL-ECU_F02_FM1***	ASIL B	Fail-safe implementation of the SW architecture and robust microcontroller	ASIL B

SW Block Element	Failure Modes	Potential Causes	Local effects	Upper-level effect	Safety level	Product Safety Mechanisms (SW-SM)	Upper-level effect with SW-SM
Front-Light Manager must (periodically 10ms) read the provided light request FLM_F01	Period too slow ***FLM_F01_FM1***	ECU overload, Task timing error	Erroneous u8IsReqValid used	***FL-SW_F02_FM1***	ASIL B	WdgM Alive Supervision 2 (10 ms)	Reset + Safe Mode

Then, the **F**ault model should be defined to mimic the occurrence of the failure mode "Period too slow". The causes of the failure mode have to be determined for our experiment. An easy example is to kill the task responsible for the execution of the function. However, for taking into account more failure mode variants, it has been decided to test different values of the period of the function. In normal behavior, the period of the runnable is 10ms. The tests have been performed with period values from 20 ms to 100 ms with a step of 10 ms.

Looking at the **A**ctivation model, two use cases (Section 5.4.5) may lead to the violation of a Safety Goal. Remember that, in the `Front-Light OS-Application`, the global variable *u8IsReValid* has an important role since it determines the setting of the headlights. As shown in Table 5.7, the local effect affecting the Front-Light Manager Module is related to this variable ("*Erroneous u8IsReqValid used*"). Indeed, the failure mode is critical when the Front-Light Manager cannot provide the new value of *u8IsReqValid*.

The corruption of the *u8IsReqValid* must be carried out when the system is in two states:

- On the one hand, the error must be injected when the headlight are already ON. In this case, the headlight may blink or switch OFF.
- On the other hand, it corresponds to a use case where the headlights are OFF, the Light Switch already ON, and an Ignition Switch command is received through the CAN.

In this last case, if the Front-Light Manager period is too long, the headlight may light with a delay.

The following **R**eadouts are needed to analyze the result of the experiment. The local effect of the FMECA line "*Erroneous u8IsReqValid used*" should be monitored in the Light Request SW-C but also the headlight state. Then, the WdgM should detect and handle correctly the error. Finally, if the

safety mechanism is efficient the headlights will be ON, otherwise they will remain OFF. This is the mode in which the system is put for safety.

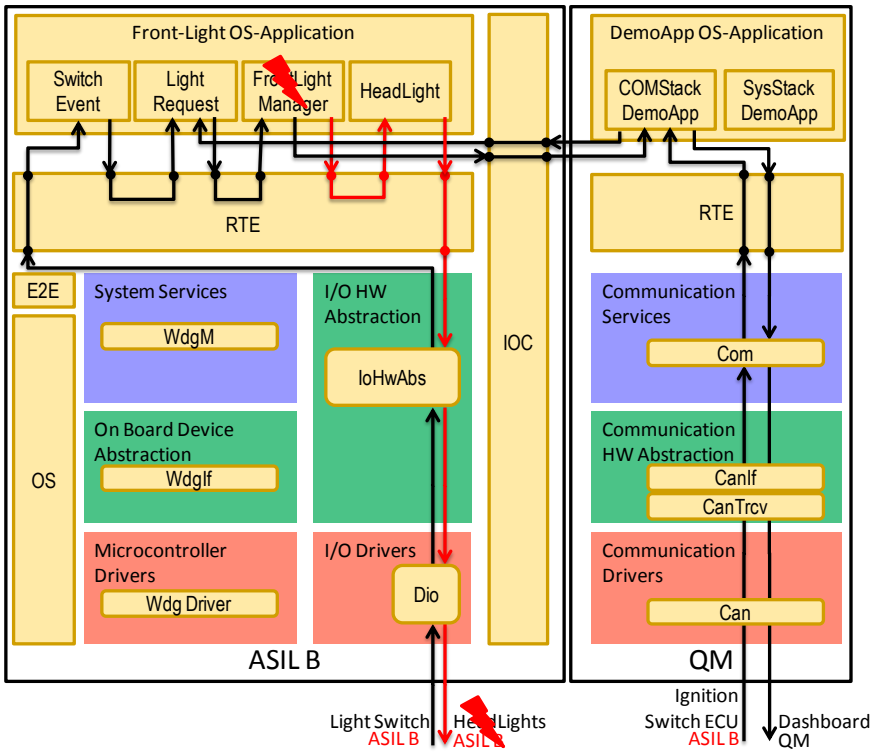


FIGURE 5.6 SOFTWARE ARCHITECTURE OF THE FRONT-LIGHT MANAGER WITH THE CRITICAL PATH IN RED OF THE SW-FMECA LINE

In this implementation, an important component for safety is the Watchdog Manager. This is why we focus on the Watchdog Manager (WdgM) (AUTOSAR-WDGM, 2014), the mechanism identified in the SW-FMECA in Table 5.6, in order to mitigate the considered SW module failure mode. The objective of the supervision of the WdgM in our application is to ensure FFI and prevent the violation of safety goals. Moreover, the WdgM is an important safety mechanism that has the same ASIL as the critical application (ASIL B in this example).

The WdgM is a generic mechanism to ensure liveness, deadline and control flow properties of an application. It is worth noting that the WdgM must be configured for a given application, in terms of deadline values, window period the Supervised Entity is alive, a graph representing the correct control flow within a given application.

To comply with the ISO 26262 requirements about the verification of the robustness of a safety mechanism, the WdgM must be analyzed. The functional behavior of the WdgM, *i.e.*, the effectiveness of the WdgM coverage, both EDC and ERC, and the characterization of error handling timing must be assessed. These tests aim at verifying that the WdgM is efficient as a safety mechanism. However, these verifications do not prevent from wrong integrations or configurations.

Due to its importance, the WdgM has been analyzed to fulfill the requirements of ASIL B. Hence, we have studied two implementations of the WdgM, a first one called “QM version” and a second one called “safety version”, which integrates safety mechanisms for improving the robustness of the implementation. We assess the robustness of the two WdgM implementations to evaluate the improvements between versions. Particularly, we assess the behavior of the WdgM, in the presence of memory

corruption (RAM/ROM/stack). This kind of experiment also evaluates the quality of the code and can highlight weaknesses in the design. The experiments have shown that corruption of memory cells may lead to the raising of false alarm, and the non-detection of liveness, deadline or control flow errors.

5.6 SW Module Level: AUTOSAR Watchdog Manager

The WdgM module is a key SW Module in AUTOSAR to ensure that the application works safely and to detect the violation of timing and logical constraints. The WdgM is part of the System Services layer and is responsible for error detection, isolation and recovery. It provides three supervision mechanisms

- Alive supervision,
- Deadline supervision,
- Control flow supervision;

and four error reactions:

- signaling errors to other AUTOSAR modules,
- logging the errors into a Diagnostic Event Manager or Development Error Tracer modules,
- partition reset: re-initialization of a specific OS-Application,
- and micro-controller reset: this will lead to a re-initialization of the MCU hardware and the complete software.

The WdgM is also responsible for the management of a watchdog driver (Wdg) of system (integrated in µC or external) via the watchdog interface (WdgIf): the watchdog driver periodically refreshes a hardware counter. Hence, if the hardware counter is not refreshed, then a software reset is triggered.

The WdgM supervisions are based on the notion of *Supervised Entities – SE*. SEs have no fixed relationship with software blocks or software modules in AUTOSAR, *e.g.*, SW-Cs, CDDs, RTE, BSW modules, *etc.* However, a SE is linked to one or several software modules implementing a functionality that needs to be monitored: alive monitoring, deadline monitoring or control flow monitoring.

Concerning its implementation, the monitoring is based on numbered checkpoints and configured transitions. A checkpoint is here defined as a step in the control flow within a SE. A SE sends a checkpoint to the WdgM (call of the WdgM API: *WdgM_CheckpointReached*) depending on its execution (start and end of an action, each step of a process, *etc.*). In the AUTOSAR WdgM terminology, a checkpoint is defined more precisely as *a point in the control flow of a Supervised Entity where the activity is reported to the Watchdog Manager*. Then, the WdgM verify that the received checkpoint is coherent with the defined supervision. These supervisions work as follow.

5.6.1 Alive Supervision

An alive supervision enables to verify that the "*SE constraints on the number of times they are executed within a given time span are respected. By means of Alive Supervision, Watchdog Manager checks periodically if the Checkpoints of a Supervised Entity have been reached within the given limits. This means that Watchdog Manager checks if a Supervised Entity is run not too frequently or not too rarely"*.

The alive supervision may filter the occurrence of a failure (too many or not enough received checkpoints). Indeed, the verification of the counter of checkpoints received during the period is done within

a range. *Counter_Min* < *CheckpointReceived* < *Counter_Max*. Although this corresponds to error detection, the alive supervision can be configured to confirm the defect during several periods before triggering a reaction.

5.6.2 Deadline Monitoring

The deadline supervision checks the timing transition between two checkpoints (start checkpoint and end checkpoint) of a SE. When the WdgM receives the `start` checkpoint, it starts a timing counter. On the reception of the `end` checkpoint, the WdgM verifies that the timing counter is in the configured bounds. If not, a reaction is triggered.

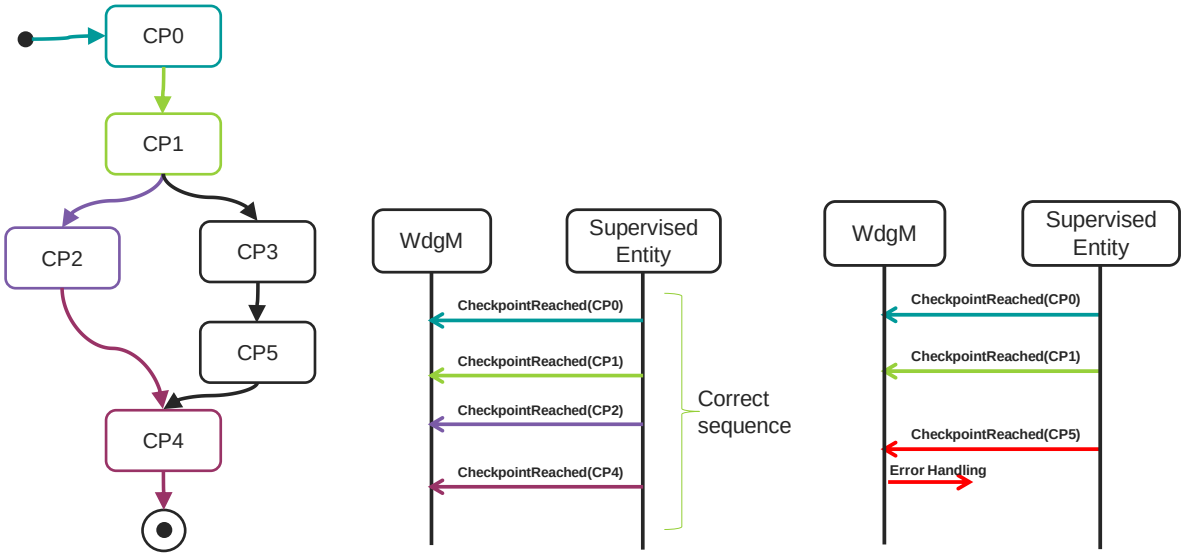
5.6.3 Control Flow Monitoring

In Control Flow monitoring, at runtime, the SEs call the WdgM API to send a checkpoint at each predefined steps of a process. The WdgM checks that the checkpoints follow a graph of the valid sequences. These graphs are defined statically during the configuration of the WdgM.

An example is shown in Figure 5.7. We assumed that a supervised entity is a task in which there are six checkpoints (CP). These checkpoints are called following two mandatory sequences (we considered that one runnable *X* sends one checkpoint *CPX*):

1. *CP0* ➔ *CP1* ➔*CP2* ➔*CP4*
2. *CP0* ➔ *CP1* ➔*CP3* ➔*CP5* ➔*CP4*

Both valid sequences are represented in the reference graph (A). In the first example given, we illustrate a valid sequence (B): the execution in the proposed sequence is correct and will not trigger a reaction. In the second example (C), we illustrate an incorrect sequence; indeed, in the second sequence, there is no transition between CP1 and CP5, then the execution flow is incorrect and the WdgM triggers a reaction.



(A) Reference graph (B) Correct control flow (C) Error detected

FIGURE 5.7 AUTOSAR WDGM: CONTROL FLOW MONITORING EXAMPLE

The overall WdgM functions and the interactions with other software modules are synthesized in Figure 5.8.

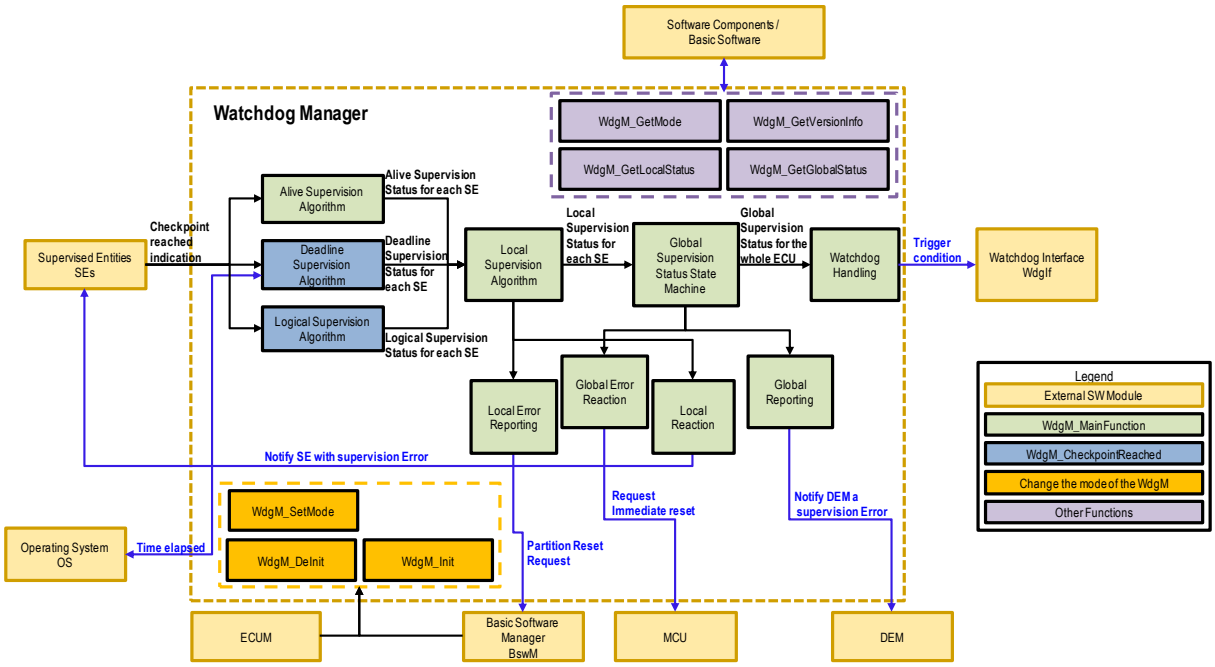


FIGURE 5.8 WDGM FUNCTIONAL DESCRIPTION

5.7 FIA at SW Module Level

We performed Safety analyses on this software module, considering it as a SEooC. We first identify the failure modes of this module. They can be categorized in:

- **False alarm**: the WdgM unexpectedly triggers a reaction (signaling an error, logging an error, resetting the target, *etc.*). It should be noted that a false alarm is not safety-critical for the system as a whole. However, it may have a bad impact on the availability of the service.
- **False negative**: it is a bad coverage of the detection or the management of the error, *i.e.*, the WdgM does not detect an erroneous behavior. The consequences of this failure can be more important. Indeed, the non-detection may lead the violation of a safety requirement in a case of double failures.
- **Timing error (too soon/too late)**: The most interesting case is when the WdgM detects and handles an error, a timing delay, which is far beyond what is expected. This is often called error detection latency.

These failure modes categories are then detailed according to the interactions with Watchdog interface WdgIf, the DEM, the MCU, the SEs and the BSWM. For example, considering the interaction with the MCU, the role of the WdgM is the following: "The WdgM must request an immediate reset of the microcontroller by calling *Mcu_PerformReset*". In this case, the considered failure modes are the following:

1. No request of immediate MCU reset (False Negative)
2. Unintended request of immediate reset (False Alarm)
3. Timing error in the request of the reset (too soon/too late)

Finally, we have identified 21 failure modes of the WdgM.

FTAs have been performed in order to find the possible causes of all WdgM failure modes. An example is given in Figure 5.9 for the case "no request of the immediate MCU reset" failure mode. We found potential causes and then proposed mechanisms. For example, there is a global variable used to store the global state of the WdgM: *WdgM_udteGlobalStatus*. The corruption of the data in this global variable is critical as it can lead to all the failure modes of the WdgM. To prevent these side effects, it has been decided to triplicate the data in different memory cells, in order to mask the error affecting one replica using, a majority voting technique.

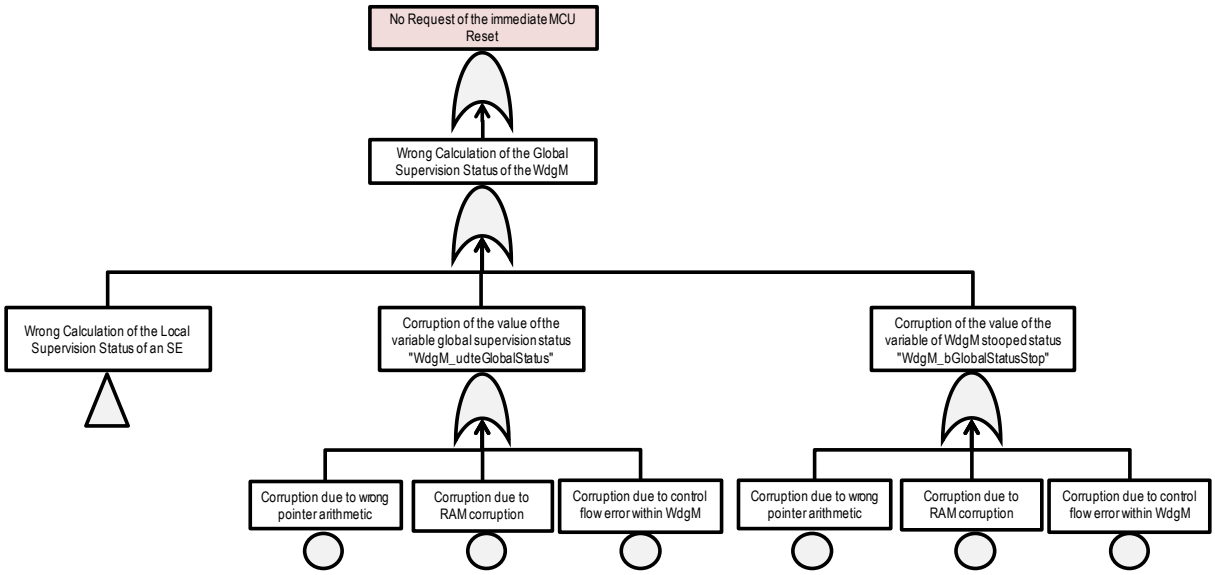


FIGURE 5.9 PARTIAL FTA OF "NO REQUEST OF THE IMMEDIATE MCU RESET" FAILURE MODE

In addition, a safety analysis has highlighted that there was a deficiency in the implementation of the QM version as some errors can remain undetected.

- Deadline Errors: the violation of a deadline may not be detected, or can be detected too late. Indeed, if the final checkpoint is not received on time, no error is detected.
- Control Flow Errors: if the application stops sending its checkpoints in the middle of the control flow graph, no error will be detected by the WdgM and the graph is incomplete.

In order to handle these cases, it was decided to change the implementation of both the deadline and the control flow supervision in order to check timing limits, whether or not the last checkpoint is received by the WdgM.

Finally, two implementations of the WdgM have been developed to illustrate the efficiency of error detection and recovery mechanisms provided by this generic module. The following instances are used in the experiments reported in next chapter:

- A QM version, which implements AUTOSAR requirements, that is limited in terms of error detection coverage, regarding deadline and control flow errors.
- A Safety version, which implements the requirements added by the safety analyses: enhancement of the robustness of the implementation and error detection deficiencies.

The fault injection experiments have been defined according to the safety analyses reported Figure 5.9

5.8 Lessons Learnt

In this chapter, we have illustrated the continuity between safety analysis and fault injection experiments. Although our case study is a simple application, we have performed analyses on a realistic system, starting from the top-level safety goals down to safety mechanisms at the software block level. Such mechanisms are the targets for the fault injection experiments. Thank to our approach, these mechanisms are linked through both S and Z chains to safety issues at upper levels, product and system. This is a core benefit of our approach: traceability in the handling of safety requirements.

Our analysis started from Undesired Events (UEs) at system level and led to the identification of safety goals together with their ASIL allocation. At product level, some elements have been identified and the FIA process was applied to precisely identify the failure modes of the corresponding functions. In this step of the product-level analysis, we have shown that some failure modes may lead to the violation of the safety goals at upper-level, *i.e.* the system level. A list of safety mechanisms was identified at product level then. Going one-step further in the analysis, *i.e.* considering SW blocks, we have shown that product level mechanisms can be related to concrete safety mechanisms or implementation choices. The WdgM is a concrete module responsible for the implementation of a collection of parameterized safety mechanisms, namely alive, deadline and control flow monitoring mechanisms. The implementation choices may also lead to some safety issues. The AUTOSAR-based implementation and the coexistence of QM and ASIL B OS application may lead to interference that are also part of the analysis at low level. FFI can be solved thanks to partitioning concepts.

Fault injection in ISO 26262 must be perceived as a causal chain through the various development and verification steps. Fault injection at the upper abstraction level can be interpreted as a “virtual” evaluation by fault injection. The target here is not concrete. Going down to more detailed levels, one can see that the target elements become more concrete. The “virtual” fault injection becomes then concrete fault injection whose aim is to evaluate the efficiency of safety mechanisms (EDC and ERC). The S-shaped causal chains trace this route from “virtual” fault injection down to concrete fault injection and by the way determine what kind of fault injection experiments must be carried out.

Controversial questions can be raised now:

Is the notion of fault injection at upper level really sound?

The ISO 26262 standards advocate fault injection has early as possible in system design. That’s a fact in the current version of the standard. A clear interpretation of this recommendation is necessary, which is the main motivation for this work. Our interpretation is that fault injection at upper design levels corresponds to detailed safety analyses, these being done using FMECA or FTA. A row in an FMECA table represents the behaviour of a given element in the presence of fault: a fault (the potential cause) may lead to a failure mode of the element having thus a local effect on its behaviour (due to an error), but also upper level effects (propagation) that must be handled by safety mechanisms (fault tolerance design patterns). This phrasing is very similar to the definition of a fault injection experiment, at least partially since measures cannot be computed at abstract levels. However, the recursion ends as soon as concrete items are found (SW or HW blocks) and then conventional fault injection experiments can be carried out to get EDC and ERC measures.

Was it possible to identify the fault injection experiments without FIA?

The answer is yes. Anyone can understand that the WdgM is a target for fault injection and that EDC and ERC measures are of high interest. The benefit of FIA relies again on the traceability of the experiments carried out regarding the upper level safety goals. Improving the traceability is essential to demonstrate the completeness of the tests carried out with respect to the system safety goals and the Undesired Events that must be avoided, closing thus the loop between FIE and FIA.

In the next Chapter, we describe the implementation of experiments carried out to illustrate the measures that can be obtained by fault injection on a real target, namely the WdgM identified in our case study.

Chapter 6 FAULT INJECTION EXPERIMENTS

6.1 Fault Injection Platform

A fault injection tool has been developed at Valeo, in order to implement the FI campaigns designs from the safety analyses. We mainly focus on the integration of two techniques: Software-Implemented Fault Injection (SWIFI) and Nexus-Based Fault Injection. These two methods have been introduced in Chapter 1.

6.1.1 Fault Injection Environment

Figure 6.1 shows an overview of the FI Environment

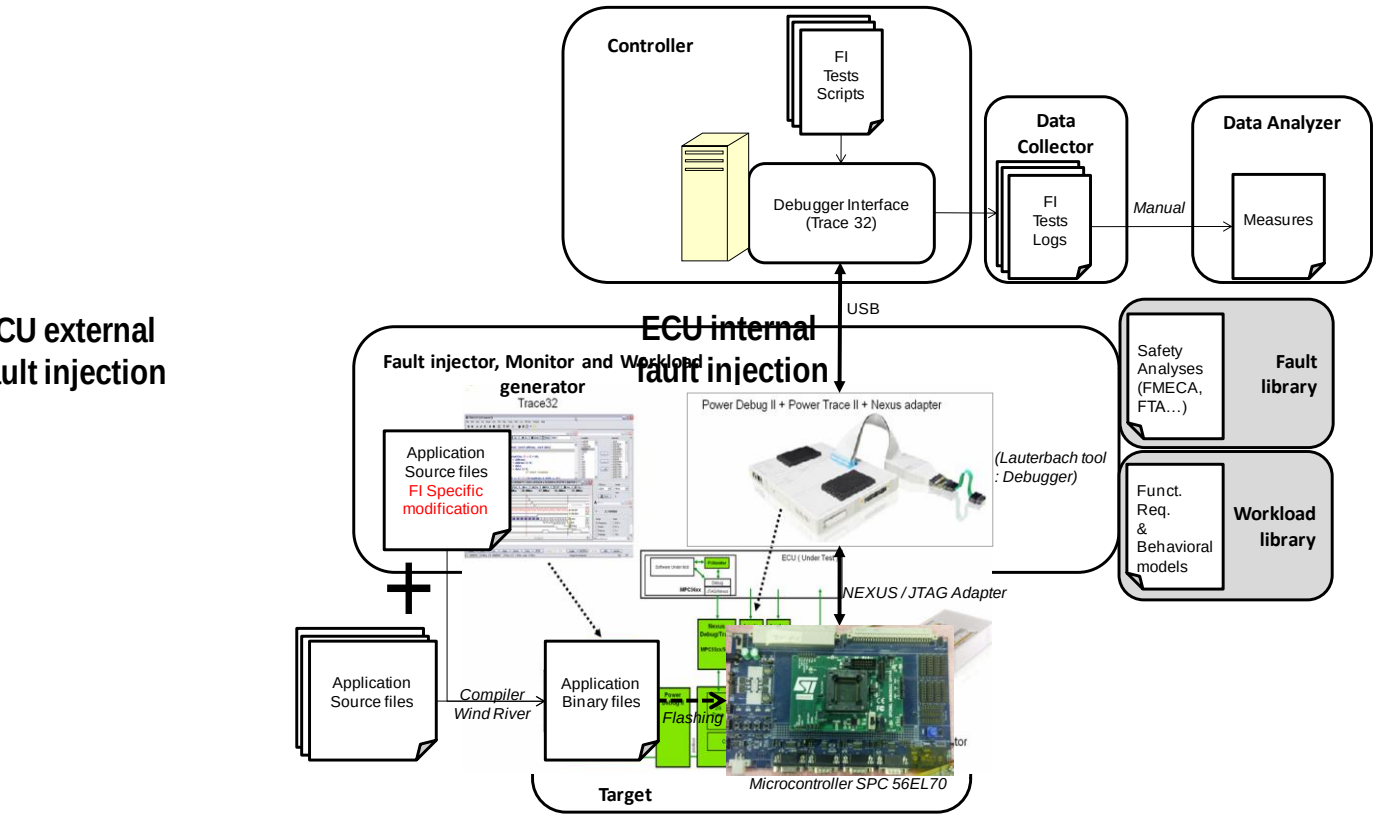


FIGURE 6.1 FAULT INJECTION ENVIRONMENT

This environment encompasses:

- The **Target System** is composed of the Front-Light Manager application (binary files ***ELF*** compiled from sources with Wind River Compiler) running on the SPC56EL70 microcontroller described before.
- The **Controller** is composed of the interface of the debugger Lauterbach TRACE 32. It enables fault injection experiments via scripts (PRACTICE Language). It also captures the execution trace of the target, controls the access to the memory for monitoring some variables, and provides commands to start the fault injection test cases according to a specified workload.
- The Lauterbach debugger (Lauterbach, 2015) is a central component of the Environment. It is connected to the controller via a USB link and to the target throughout a Nexus Probe. The debugger provides access to all memory sections of the microcontroller, particularly by monitoring the trace of the execution of the application (**Monitor**).

- The Lauterbach debugger is also a **Fault Injector**, as it allows corrupting/modifying memory locations of the application. These capabilities are provided by the implementation of Nexus Class 3+ defined by the Nexus 5001 Forum™ (Nexus5001, 2015). This class of debugger enables two important features to perform read/write into the memory without any impact on the real-time execution of the application. This "On-the-fly" runtime memory access does not add temporal overhead. Moreover, the debugger also takes advantages of on-chip watch points (a watch point enables the activation of the fault injection experiments to be triggered or signaling application events without stopping the application). Finally, the debugger enables monitoring the behavior of the execution of the Software through the memory to collect observation data (readouts) and to synchronize the activation of the experiments (**Workload Generator**), and finally, modifying the memory to inject fault in memory, registers, *etc.* (**Fault Injector**)
- We also use SWIFI method to inject specific fault/error/failure. Hence, we instrument the code to mimic the faulty behavior directly by modifying the source code of the faulty software module, and finaly compiling the mutant application.
- The **Data Collector** stores the data collected during the experiments into log files. These are timing information, variables values and events.
- The **Data Analyzer** in our environment is mostly manual. The data is gathered automatically in the logs, however the analysis of the results has to be done manually, *e.g.*, the categorization of the experiments.

6.1.2 Fault Injection Characterization of the Tool

We present the characterization of the fault injection techniques integrated in tool by considering the following properties, defined in (Arlat, et al., 2003):

- ***Reachability***: Using both pre-runtime SWIFI and Nexus-based fault injection, we are able to manage a high level of reachability. Indeed, we are able to inject fault directly in the memory, the CPU registers using the debugger. It is also possible using SWIFI to corrupt higher levels of granularity, by injecting information explicitely processed by the computing system.
- ***Controllability***, with respect to space and time: The controllability is also high as the fault can be triggered using low-level mechanisms from the debuggers (break and watch points in the program flow, writing/reading access to specific data). It may also be used together with SWIFI, for instance to trigger a branch of instrumented code.
- ***Repeatability*** (with respect to experiments) and ***Reproductibility*** (with respect to results): A high repeatability is attained thanks to the high level of controllability. However, a distributed architecture with complex environment (multiple ECU on the network) may be more difficult to synchronize and thus, it would be more difficult to ensure repeadability in this case. In our case the microcontroller behavior allows a high repeatability
- ***Non-intrusiveness***: The intrusiveness of the Nexus capabilities is very low. The intrusion can also be related to the use of watch points, but there are few "real-time" watch points in practice. In addition considering the SWIFI, it is clear that the intrusiveness depends on the size of the corruption made in the source code. In our case, this second type of intrusion is negligible. Then, the use of SWIFI does not modify the behaviour or the structure of the safety mechanisms. Otherwise, the experiment results may be biased.
- ***Time measurement*** (*e.g.*, error detection latency): Here, time measurements are performed by the debugger. It is possible to get the execution time between two events.
- ***Efficacy*** to generate significant experiments: Generally, this property aims at characterizing a fault injection technique together with a fault model. In our approach, the efficacy is based on the identification of the tests cases, more precisely the selection of the fault to be injected.

To conclude, the tool enables handling most of the Fault Injection experiments on any target application running on any micro-controller.

6.2 WdgM Implementations Assessment

In the whole section, we consider two implementations of the WdgM. The QM version has been developed without specific mechanisms, and the so-called safety version that integrates these mechanisms and additional improvements. The campaign described in this section does not run the Front-Light Application. The software architecture involves another SW-C: the *ComStackDemoApp*, especially designed to generate the workload.

6.2.1 Error Detection and Error Recovery Coverage

Firstly, we have tested if the implementation was efficient to detect and recover from Alive / Deadline / Control Flow errors (in other words, the aim of these tests is to verify that the WdgM fulfills its functional specification when integrated).

The measures and the target have been previously defined, the objective being the verification of the effectiveness of the EDC and ERC of the WdgM. Another aspect is the verification that the weaknesses identified in the QM implementation are well covered in the safety version of the WdgM. Now we will focus on the definition of the fault model, the Activation model and the Readouts.

The considered ***Fault Model*** corresponds to a wrong behavior of a SE, particularly the *ComStackDemoApp*. Two faulty SE behaviors have been used: too many checkpoints sent by the SE or not enough.

For the deadline supervision, the following faults are considered:

- Reception of the end checkpoint first from the SE
- Reception of the start checkpoint only from the SE
- Reception of the end checkpoint too late from the SE. We made several experiments in which we increase the sending instant of the end checkpoint by 5 ms at each experiment.

Concerning the control flow monitoring we use a reference graph as an oracle. This graph corresponds to the correct behaviour of the application and it is established a priori. We have injected all the possible sequences without repetition. Hence all the sequences that do not comply with the graph must be detected.

For the ***Activation model***, we do not identify a specific use case. However, we have to make sure that the initialization of the application is correct and completed before starting the experiment.

The objective is to evaluate ***Measures***, *i.e.* the coverage of the error detection mechanisms implemented in the WdgM and the identification of the corresponding failures. To this aim, we have to monitor the local status variable of each SE, a sort of flag indicating if an error was detected by the WdgM (Detection Coverage).

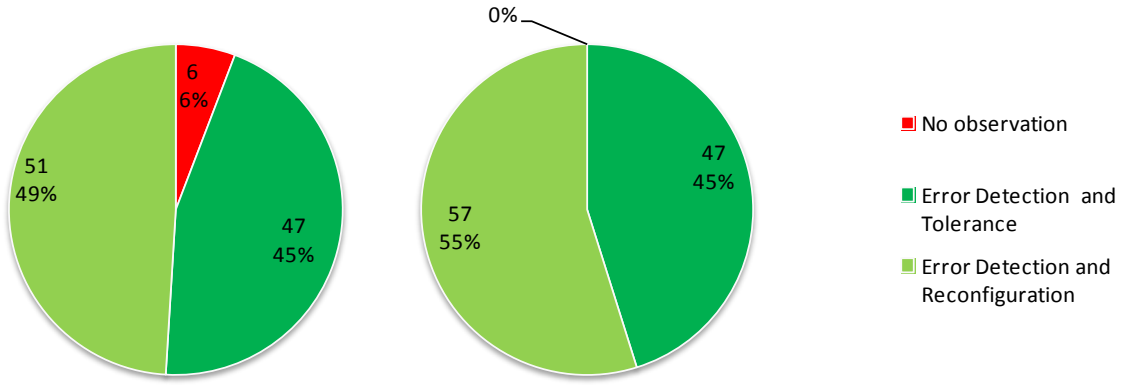
Then, we have to monitor the reaction of the WdgM (Recovery Coverage). In this case, we configured the WdgM in order to perform an immediate reset by calling the *Mcu_PerformReset* function. The call must be monitored as well as the reset (we put a watch point on the *main()* function of the application

to trace the resets). The reset corresponds to the recovery action, *i.e.*, the WdgM performed correctly its recovery action.

104 different experiments were carried out in the fault injection campaign, and we obtained the results presented in Figure 6.2. The WdgM QM version detects 94% of the errors. Among the detected errors, 49% lead to reset the application, and 45% lead to no reaction.

About 6% of the fault injection experiments led to undetected errors, *i.e.* no observation. These 6% of undetected error are of prime importance. They correspond to experiments highlighted in the Safety Analyses. For example, when an end checkpoint is never received by the WdgM, the error (control flow or deadline monitoring) is not detected.

We have performed the same set of experiments for both the QM and the Safety implementation of the WdgM. The objective is the verification of added features (timeout for the detection of deadline errors and for the detection of a correct incomplete control flow sequence) and the evaluation of their efficiency. We can observe in Figure 6.2, that the detection coverage of the Safety version is 100% with the same experiments. It is worth noting that the previously undetected errors are now detected and then the expected reaction is performed.



(A) WdgM QM Version (B) WdgM Safety Version

FIGURE 6.2 EFFECTIVENESS OF EDC/ERC OF THE TWO WDGM IMPLEMENTATIONS (104 EXPERIMENTS)

We can conclude that this campaign improves our confidence in the error detection and Error Recovery coverage of the WdgM against alive errors /deadline errors /control flow monitoring. We also validate a solution, which take into account errors that are introduced by the implementation of functional behavior of the WdgM.

6.2.2 Timing Evaluation of the WdgM

In parallel to the evaluation of both the EDC and the ERC, we also evaluate the timing behavior of the corresponding error detection and recovery mechanisms. The objectives of these experiments are to measure and evaluate the error detection time and the reaction time of the WdgM with regard to alive, deadline and control flow monitoring.

According to the configuration of the WdgM that has been made, the period of execution of the main function of the WdgM is 20 ms. Then, concerning the Deadline and control flow monitoring, we considered that the error is detected after the execution of the *WdgM_MainFunction().*

Moreover, the alive monitoring filters three failed periods before signaling the error. Then, in the worst case, an alive error is detected in less than four periods, *i.e.*, 80 ms.

We obtained results summarized in Table 6.1. The detection time of both alive monitoring and control flow monitoring are within the range of the configuration. Considering the deadline monitoring for the QM implementation, the detection time depends on the failure type. If the "end checkpoint" is received very late, the detection will be as late as the reception of this checkpoint. In our tests cases, all the detection delays were greater than 19 ms for the QM version. The detection delay has been improved in the safety version. Indeed, the detection is also verified in the periodic main function of the WdgM by verifying if the deadline is exceeded. In our case, the safety module detects the error between 14 ms and 20 ms. In the safety version of the WdgM, it is possible to set a time bound for the detection.

TABLE 6.1 RESULT OF TIMING CHARACTERIZATION OF THE WDGM

Supervision	**Alive monitoring**		**Deadline monitoring**		**Control flow monitoring**	
WdgM version	**QM**	**Safety**	**QM**	**Safety**	**QM**	**Safety**
Detection Time	**~63,9 ms**	**~62,7 ms**	**19 ms < Detection time**	**14 ms < Detection time <20 ms**	**~8.4 ms**	**~8.7 ms**
Reaction time (Mean)	**~ 36,8 ms**	**~ 27,7 ms**	**~ 37,7 ms**	**~ 27,1 ms**	**~ 36,4 ms**	**~ 26,5 ms**

Then we also measure the reaction time. Here, we consider that the reaction is the time between the detection and the start of the main function of the application after the reset. The reaction time is stable in the two versions. However, the implementation in the safety version is 10 ms faster.

Even if the absolute values are not significant for all the configurations of the WdgM, the result shows that the safety version improves the QM version from both coverage and timing viewpoint.

6.2.3 Robustness of the implementation of the WdgM

Finally, we have to evaluate the robustness of the WdgM implementation against interferences from HW faults (bit-flip in the memory/registers), and from others software modules malfunctions during runtime.

This assessment is important as we considered the AUTOSAR WdgM as a Component Off-The-Shelf (COTS). This module can be reused on a different architecture, and thus a robust implementation should be made and assessed, especially when the WdgM will be used in highly safety critical system.

It is interesting to note that the memory-related interferences in the WdgM do not impact the safety of the System; such type of interference leads to false alarms and a reset. However, interferences may be safety relevant with second order cut sets. In this case, no detection of deadline, alive or control flow errors, may violate a safety goal.

All the faults have been defined using the Z-Shaped causal chain. Indeed, the fault model used for this campaign is the corruption of variables of the WdgM, these variables have been identified as safety critical in FTA performed in order to find causes of the WdgM failure modes. Then, for each identi-

fied variables, we inject errors following a data type fault model. Basically, the injected values are based on the data type of the variable, *e.g.*, the global status of the WdgM is coded into the variable *WdgM_udteGlobalStatus* which type is 8 bits unsigned integer, according to the AUTOSAR specification. Moreover, there are five valid values from zero to four (see Table 6.2). To corrupt the variable, we forced the value of *WdgM_udteGlobalStatus* to all the valid inputs and we add the maximum value of the type $N\_MAX\_UINT8 = 2^8 - 1 = 255$, and a median variable $N\_MED\_UINT8 = 2^8/2 = 128$.

TABLE 6.2 AUTOSAR SPECIFICATION OF *WDGM_GLOBALSTATUSTYPE* (AUTOSAR-WDGM, 2014)

Name:	WdgM_GlobalStatusType		
Type:	Uint8		
Range:	WDGM_GLOBAL_STATUS_OK	0	Supervision did not show any failures.
	WDGM_GLOBAL_STATUS_FAILED	1	Supervision has failed but is still within the limit of allowed failures.
	WDGM_GLOBAL_STATUS_EXPIRED	2	Supervision has failed, the allowed limit of failures has been exceeded, but the Watchdog Driver has not yet been instructed to stop triggering.
	WDGM_GLOBAL_STATUS_STOPPED	3	Supervision has failed, the allowed limit of failures has been exceeded, but the Watchdog Driver has been instructed to stop triggering. A watchdog reset is about to happen.
	WDGM_GLOBAL_STATUS_DEACTIVATED	4	WdgM is not initialized and therefore will not manage the watchdogs.
Description:	This type shall be used for variables that represent the global supervision status of the Watchdog Manager module		

This type of corruption has been done with all the variables identified in the FTA from Figure 5.9.

Considering the activation model, we have shown in section 5.4 that there are two main cases where a corruption of the WdgM may lead to the identified failure modes:

- ***Activation 1:*** In normal mode, *i.e.*, in the use case the supervised entities work as defined; they send checkpoints regularly for alive supervision, within deadlines and following the control flow graph. In this use case, the considered fault model may lead to false alarms.
- ***Activation 2:*** In the second use case, we consider that an error is present in one supervised entity. Then, this corruption may remain undetected by the WdgM.

To correctly implement these two cases, we have to make sure that the corruption won't be overwritten during the SE execution.

Concerning the readouts, we have to monitor the location of corrupted data by the fault injection. Then, we monitor the entire set of variables that flag the error detected by the WdgM (WdgM global status, local status, *etc.*). Then a reaction should be called by the WdgM: immediate reconfiguration though *WdgM_PerformReset* and the reset should be monitored. In this campaign, the reset could be due to the reaction of the WdgM after the detection of an error or because of the software watchdog is not periodically kicked by the WdgM.

It is important to mention that we did not evaluate timing issues in this campaign. The robustness may be even less when considering timing performance of the detection (error detection latency).

In this campaign, we have performed 217 experiments on each version.

The experiments have been categorized as follows:

- "**No deviation Observed**" corresponds to tests where the behavior of the WdgM is identical with or without the injected fault. (case α and δ of Table 4.1).
    - the corrupted variable is refreshed with the correct value.
    - In the case of ***Activation 2***, *i.e.* in presence of a SE failure, it may correspond to experiment where a false alarm has been raised.
    - this also corresponds to test cases where the corrupted value is equal to the current value.
- "**Transient Internal Deviation Observed**" corresponds to experiments where the error propagates but is tolerated before leading to a failure mode of the WdgM. (case α and δ of Table 4.1). In fact, this case corresponds to experiments where the corruption leads to observe a deviation of the WdgM behavior (*e.g.*, unintended detection of an error in ***Activation 1*** case), but there is no effect on the WdgM outputs. Here, the detection is tolerated by the WdgM and there are no reaction called
- "**Internal latent error**" corresponds to experiments where no WdgM failure mode is observed (false alarm or false positive). However, the corruption done or the propagation of this error remain latent and is not activated by the activation of the target. (case α and δ of Table 4.1). It also corresponds to experiments where the corrupted variables are neither read nor written. Latent errors highly depend on the activation of the target.
- "**Failure mode reached**". Here, a false alarm or a false negative is observed (case β and γ of Table 4.1).

The results of the campaign on the two implementation of the WdgM are presented in the Figure 6.3

(A) WdgM QM Version (B) WdgM Safety Version

FIGURE 6.3 ROBUSTNESS CAMPAIGN ON THE WDGM IMPLEMENTATIONS (217 EXPERIMENTS)

First, a good result is that among the 214 tests cases, only 35% are non-significant tests cases (no observation). This shows that, in this particular case, the use of safety analyses in the determination of experiment lead to efficient experiments, which is a major issue, in the industry and in the domain. Moreover, we found that 35% of the experiments led to a failure mode of the WdgM. Particularly, among the 76 experiments (with the QM version) that lead to a failure mode, alive/deadline/Control flow error, have not been detected in **19 experiments** (False negative) and **57 experiments** leads to unintended resets of the WdgM.

Then, we can compare the two versions of the WdgM. First, we can easily observe that even if the number of experiments leading to a failure mode has been reduced, the result is not significant.

This shows that the proposed robustness mechanisms are either not efficient or not well implemented. In fact, in our case, the tested version implements few prescribed mechanisms. Then we have shown that the coverage of the WdgM has been improved, but the robustness of the implementation remains similar.

The results highlight that some efforts can be done on the implementation. A decision can be made to improve or not the implementation. A tradeoff has to be made between memory consumption, performances and WdgM implementation of safety requirements.

The improvement should focus on basic events that lead to Failure Modes (red) and errors that remain latent (orange).

To conclude, we have illustrated all FI experiments that should be made according to the ISO 26262 on a software safety mechanism:

1. *demonstration of the effectiveness of the safety mechanisms*.
    a. verification of the error detection and error recovery coverage
    b. verification of timing requirements of the mechanisms
2. verification of the robustness of the implementation of the safety mechanism

6.3 Front-Light Software Verification

In the previous section we have shown that the safety mechanisms have been characterized. Now, we will verify the correct implementation of the safety requirements of the front light software. In other words, the objective is the verification of the S-shaped causal chain identified in the FMECA in Table 5.5

6.3.1 Verification of one Line of FMECA: S-Shaped Verification

18 tests cases have been performed for the considered critical path: 9 for each use cases. The results could be categorized in (see Figure 6.4):

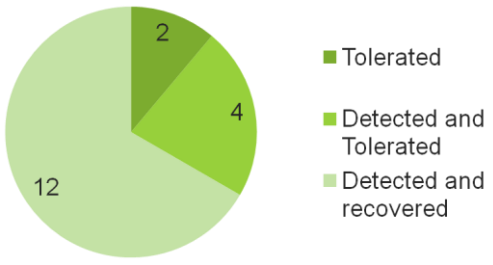


FIGURE 6.4 VERIFICATION OF ONE FMECA LINE (18 EXPERIMENTS)

- Tolerated errors: the WdgM does not detect error (according to his configuration) and the system works even if the period is partially degraded (the output is refreshed less than specified). [period = 20 ms]
- Detected errors and no reaction: the WdgM detects the error but does not perform any recovery actions. The system works in a degraded mode (the output is refreshed less than specified). [period = 30 ms or 40 ms]

- Detected and recovered errors: in these cases, the WdgM detects the alive error, and start a reset of the microcontroller. Then, in use case 1, there is a blinking where the headlight are OFF during 37 ms (reset time) and then the system works properly (headlights are ON). In use case 2, *i.e.* the headlights are OFF and the user requests to switch them ON. The system will not switch the headlights ON before a delay of 220 ms because of the injected error, that corresponds to the WdgM detection time and reset of the application. [Periods between 50 ms and 100 ms].

All the tests could be put in cases α and δ of the Table 4.1.There is no violation of SG1 or SG2.

To conclude, the implemented application enables to meet the requirements and the safety mechanisms handle correctly the faulty behavior within a bounded response time. Here, a single failure is not sufficient to violate a safety goal alone, in the worst case, the failure mode occurs but a reset is triggered fast enough to remain unnoticed.

6.3.2 Global Verification of the FMECA Spreadsheet

Finally, we consider the evaluation of the software architecture, and the impacts of all the software faults on the system. The objective is to verify that there are no violations of the safety goals and that the safety mechanisms proposed in the FIA process are efficient.

As we saw in Section 5.4.5, the complete FMECA spreadsheet encompasses 48 lines that may violate the safety goal: SG1. For each failure mode, we defined a set of experiments similarly to the line considered in Section 5.5. The campaign is composed of 218 experiments.

These failure modes have been injected using SWIFI: modification of the RTE module to control the flow of the executed runnables, and modification of the memory using the debugger, in order to modify several variables **spuriously** or **permanently**.

The result of the campaign is given in the left pie chart of Figure 6.5.

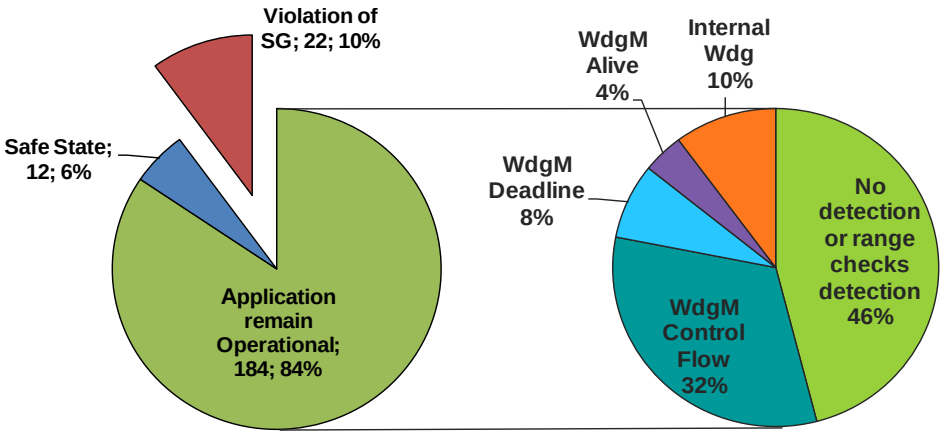


FIGURE 6.5 GLOBAL RESULT OF THE VIOLATION OF THE SAFETY REQUIREMENTS AND THE TRIGGERED SAFETY MECHANISMS (218 EXPERIMENTS)

First, it should be noted that experiments have been categorized into three types:

Application remains operational: These are the experiments which lead to "No observation", or "Error detected and/or reaction". At the end of the experiment, the application is operational.

Safe state reached: This category encompassed the experiments that lead to multiple resets of the target. In this case, this mainly encompassed permanent error corruption of critical variables. Then, it leads to multiple reset of the target. This behavior is handled by a mechanism that stops the application after three consecutive resets. The loss of the functionality is considered here as safe.

Violation of a safety goal: It corresponds to the blinking of the application or the loss of the headlights while they should be ON. In fact, we found that these experiments correspond to the line of the FMECA where a lack of coverage has been identified. The cause of this problem is the non-detection of "valid errors"; a valid error should be understood as ' an incorrect OFF value has been sent instead of a correct ON value'. However, the value is incorrect with respect to the system state. All other errors correspond to incorrect values that could be given to the ON/OFF variable (different from ON and OFF coded value) and thus easier to detect. In summary, the non-detection of permanent valid error is a key issue.

To conclude the global coverage of the safety requirement is 90%, *i.e.*, the system is safe in 90% of fault injection experiments.

Then, we show that the error detection mechanisms prevent the violation of a safety goal. The results are shown in the right pie chart of the Figure 6.5. The following categories have been found:

No detection: This category encompasses several types of experiments. First, it may correspond to errors that are tolerated by the functional behavior of the application (periodically refreshed values). It also corresponds to the errors detected by checking a range of correct values, but reported.

Then, other categories correspond to the proposed safety mechanisms:

- WdgM alive supervision of the runnables,
- **WdgM Deadline monitoring** of the execution of the critical runnables, and
- WdgM control flow monitoring.
- Finally, the **internal watchdog** is able to detect an infinite loop. In this case, the internal watchdog is responsible for the reset.

To conclude, this campaign intends to demonstrate that the safety requirements have been correctly handled. The measures show the proposed architecture handles most of the failure modes defined in the FIA.

Some problems have been found in the implementation of the front light manager. First, a wrong implementation of a range checking has been found leading to the violation of a safety goal. Then, in other experiments, a lack of coverage has been identified early in the FIA, *i.e.* some situations where a safety mechanism was not proposed in the FIA.

We have found that, in some cases, the violation of the safety requirement only appears when there is a permanent corruption of the critical value with a specific value. Improvements/modifications should then be proposed to handle these remaining issues.

6.4 Conclusion

A proof of concept of the method has been realized on simple automotive application: Front-Light System. The software architecture of this case study is based on AUTOSAR 4.X. The system has been described at different levels of development together with safety analyses, to show the traceability of

the requirements and their importance in the assessment of FI measures. At the end of the FIA process applied to this example, we have identified safety mechanisms (WdgM) that are targets of fault injection experiments.

In order to carry out the planned tests, we have developed a FI test platform. Two implementations of the WdgM (QM and safety) have been evaluated by fault injection experiments.

A first interesting result obtained with the experimental measures is the efficiency of the injected faults, *i.e.* the fact that all injected faults lead either to an error-detection or to the violation of safety properties. The term efficiency may be misleading here, but it enables the error detection coverage of the various implementations of the WdgM to be estimated. A non detection leads to the violation of a safety property and that in turn requires to improve the WdgM.

In other words, the injected faults have an impact on the target (they lead to a failure mode or they triggered a safety mechanism).

The obtained measures provide reasonable insights to demonstrate the effectiveness of the safety mechanisms (the WdgM), the correct implementation of the safety requirements, and the FFI. Our current work aims at obtaining global measures from FI experiments and optimizing the whole development process by defining an optimal set of experiments.

To conclude, this method offers interesting results for the integration of the FI in an automotive development process following the requirements of the ISO 26262. However, this may lead to significant efforts and timing overhead on a complete architecture. Hence, the FI experiments must be carefully selected.

Conclusion/Perspectives

The development of dependable systems has always been a challenge for engineers. With the growing criticality of functions allocated to E/E systems, it is becoming increasingly important to guarantee that the safety requirements have been correctly handled during the whole development process. Therefore, it is essential to propose a structured and systematic validation process that provides convincing evidences of their correct design and implementation. The work presented in this thesis is a contribution towards the satisfaction of this need.

This means that safety issues must be correctly addressed at early stages of the system design, then through all development phases down to the implementation and then intensively tested. Safety analyses are very important since they identify potential causes of failures, their effect on the system in order to identify, early in the development process, the required safety mechanisms to add in the design. Any omission at this stage means that, although the system is correctly developed, some failures may lead to undesirable effects, some being critical, i.e. potentially leading to a hazard.

Testing is a challenge as far as critical systems are concerned. Any improvement of testing methods regarding safety critical systems is of course of interest. Fault injection is a technique that complements functional testing since it focuses on the behavior of a system or a component in the presence of faults. Its contribution is well known when targeting dependability mechanisms: verification of safety mechanisms and evaluation of component robustness. Today, an open question relates to the contribution of fault injection when applied during the design phase.

More generally, the question tackled in this thesis is the complementarity between safety mechanisms definition during the design and their practical evaluation by fault injection when implemented. To our mind, this is an underlying question raised by the introduction of fault injection in the ISO 26262 standard, and this is a challenge.

The main contribution of this thesis concerns the integration of fault injection in the whole development process. Fault injection was seen as an efficient method to assess the robustness of an implemented target. Based on a method designed for experimental validation (FARM), we demonstrate that the introduction of fault injection analyses was of interest for the overall development process. We have shown that fault injection analysis is strongly related to detailed safety analyses: more specifical-

ly detailed FMECA. This analogy and its justification is a first contribution of the thesis that clarifies the meaning of fault injection at early stages of the design.

The outcome of this work is that a process based on fault injection can be applied and advantageously integrated in a standard development process for automotive systems. Fault injection analyses as described in this thesis enrich the already existing safety analyses.

Our contribution shows the major role of fault injection in the verification and validation of safety, during the whole development process.

The paradigm of S- and Z-shaped causal chains is of a great help for analyzing error propagation between the various development levels and entities. It is not only important during the design phase, but also for fault injection on an implemented target. We have in particular demonstrated the importance of the traceability of the fault propagation, using these causal chains. The first advantage of FIA is the traceability of the fault injection experiments, since fault injection experiments are defined from the FIA (S-chain), and conversely that the results of an experiment can be related to upper level items and obligations in the design (reported in the FMECA spreadsheets). This work is a clear contribution to the understanding of fault injection in the development process, since it is easier to demonstrate that a safety requirement has been tested correctly. In addition, it is of major interest in the planning of FI campaigns, as the concrete measures to assess can be identified early in the development process. The second advantage is more practical for the testing team. The fault propagation enables the identification of potential causes of a hazard to be injected into the target (Z-chain).

The proposed approach is of course compliant with the ISO 26262 standard. Indeed, safety analyses are required for all safety critical elements, and we have highlighted the importance of fault injection at all stages making a clear link between conventional fault injection on concrete targets and safety analyses. Even if the fault injection activities imply efforts, our approach moderates the impact, by reusing safety analyses, which are already performed. Now, these analyses will be used as inputs of the fault injection campaigns.

From a practical point of view, a fault injection tool have been developed, which helped us to illustrate successfully our approach on a simple example. Now, the objective is the development of this prototype, FIP, to enable the verification and validation of several targets. The next generation needs to be easier to use, and with automatic assistance in the definition of the experiments. Finally, we expect that a mature tool will be used within Valeo.

We are aware that the systematic application of our approach in the design of a system is time consuming and requires lot of efforts, particularly for performing detailed FMECA on software architectures or HW parts. However, safety analysis is highly recommended by the standard, so this can be managed. If analyses are correctly performed from the early development phases, then the experimental part will be easier.

The scalability of the approach has not been demonstrated in this work, just proofs of concepts have been done using two case studies. This is certainly something that should be done in a sizable industrial project. The first perspective of our work is thus its application to real systems.

To take advantage of this systematic approach, it is clear that the industrialization of the methodology in tools is mandatory. For example, the outputs from the safety analyses should be investigated more precisely, particularly regarding the fault model. Notably, the definition of a standardized fault model

can help choosing the most appropriate technique (or tool) for performing the fault injection experiments. This problem is strategic in a project, as the tests means should be identified at the beginning of a project. Another aspect is to use the tools to follow the S- and Z-chains and to look for completeness of the experiments. The omission of a failure mode is still an open problem, but our approach limits this drawback.

A second perspective is the definition of benchmarks for COTS. We partially illustrate this issue with the WdgM, as it can be reused in several applications. Generally, this is particularly of interest for AUTOSAR architecture. Similar campaign can be performed with different implementation of the same WdgM module, but parameterized differently. The campaign, based on the safety analyses of the COTS, leads to define a set of experiments that could be performed on different implementation of the COTS. This approach is of prime importance in the case of safety mechanisms. This will help defining the most appropriate version of the component, i.e., which mitigates the faults the more efficiently.

Finally, it is worth noting that the proposed approach can be applied to any critical system in many domains: railways, avionics, medical devices (Park, Yi, Kwon, & Jeon, 2014), *etc.*

APPENDIX 1

Analyzed Faults or Failures Modes in the Derivation of Diagnostic Coverage (ISO 26262, Part 5, Annex D)

Element		Analysed failure modes for 60 %/99%		
		Low (60 %)	**Medium (90 %)**	**High (99 %)**
General elements				
E.E Systems		No generic fault model available. Detailed analysis necessary.	No generic fault model available. Detailed analysis necessary.	No generic fault model available. Detailed analysis necessary.
Electrical elements				
Relays		Does not energize or de-energize. Welded contacts.	Does not energize or de-energize. Individual contacts welded	Does not energize or de-energize. Individual contacts welded
Hamesses including splice and connectors		Open Circuit Short Circuit to Ground	Open Circuit Short Circuit to Ground (d.c Coupled) Short Circuit to Vbat Short Circuit between neighbouring pins	Open Circuit Contact Resistance Short Circuit to Ground (d.c coupled) Short Circuit to Vbat Short Circuit between neighbouring pins Resistive drift between pins
Sensors including signal switches		No generic fault model available. Detailed analysis necessary. Typical failure modes to be covered include. Out-of-range Stuck in range	No generic fault model available. Detailed analysis necessary. Typical failure modes to be covered include. Out-of-range Offsets Stuck in range	No generic fault model available. Detailed analysis necessary. Typical failure modes to be covered include Out-of-range Offsets Stuck in range Oscillations
Final elements (actuators, lamps, buzzers, screen)		No generic fault model available. Detailed analysis necessary.	No generic fault model available. Detailed analysis necessary.	No generic fault model available. Detailed analysis necessary.
General semiconductor elements				
Power supply		Under and over Voltage	Drift Under and over Voltage	Drift and oscillation Under and over Voltage Power spikes
Clock		Stuck-at[a]	d.c. fault model[b]	d.c. fault model[b] Incorrect frequency Period jitter
Non-volatile memory		Stuck-at[a] for data and addresses and control interface, lines and logic	d.c. fault model[b] for data and addresses (includes address lines within same block) and control interface, lines and logic	d.c. fault model[b] for data, addresses (includes address lines within same block) and control interface, lines and logic
Volatile memory		Stuck-at[a] for data, addresses and control interface, lines and logic	d.c. fault model[b] for data, addresses (includes address lines within same block and inability to write to cell) and control interface, lines and logic. Soft error model[c] for bit cells	d.c. fault model[b] for data, addresses (includes address lines within same block and inability to write to cell) and control interface, lines and logic Soft error model[c] for bit cells
Digital I/O		Stuck-at[a] (including signal lines outside of the microcontroller)	d.c. fault model[b] (including signal lines outside of the microcontroller)	d.c. fault model[b] (including signal lines outside of the microcontroller) Drift and oscillation
Analogue I/O		Stuck-at[a] (including signal lines outside of the microcontroller)	d.c. fault model[b] (including signal lines outside of the microcontroller) Drift and oscillation	d.c. fault model[b] (including signal lines outside of the microcontroller) Drift and oscillation
Specific semiconductor elements				
Processing units	**ALU - Data Path**	Stuck-at[a]	Stuck-at[a] at gate level	d.c. fault model[b] Soft error model[c] (for sequential parts
	Registers (general purpose registers bank, DMA transfer registers…), internal RAM	Stuck-at[a]	Stuck-at[a] at gate level Soft error model[c]	d.c. fault model[b] including no, wrong or multiple addressing of registers Soft error modelc
	Address calculation (Load/Store Unit, DMA addressing logic,	Stuck-at[a]	Stuck-at[a] at gate level Soft error model[c] (for	d.c. fault model[b] including no, wrong or multiple addressing

Element		Analysed failure modes for 60 %/99%		
		Low (60 %)	**Medium (90 %)**	**High (99 %)**
	memory and bus interfaces)		sequential parts)	Soft error model[c] (for sequential parts)
	Interrupt handling	Omission of or continuous interrupts	Omission of or continuous interrupts Incorrect interrupt executed	Omission of or continuous interrupts Incorrect interrupt executed Wrong priority Slow or interfered interrupt handling causing missed or delayed interrupts service
	Control logic (Sequencer, coding and execution logic including flag registers and stack control)	No code execution Execution too slow Stack overflow/underflow	Wrong coding or no execution Execution too slow Stack overflow/underflow	Wrong coding, wrong or no execution Execution out of order Execution too fast or too slow Stack overflow/underflow
	Configuration Registers	_	Stuck-at[a] wrong value	Corruption of registers (soft errors) Stuck-at[a] fault model
	Other sub-elements not belonging to previous classes	Stuck-at[a]	Stuck-at[a] at gate level	d.c. fault model[b] Soft error model[c] (for sequential part)
Communication	**On-chip communication including bus-arbitration**	Stuck-at[a] (data, control, address and arbitration signals)	d.c. fault model[b] (data, control, address and arbitration signals) Time out No or continuous arbitration	d.c. fault model[b] (data, control, address and arbitration signals) Time out No or continuous or wrong arbitration Soft errors (for sequential part)
	Data transmission (to be analyzed With ISO 26262-6:2011, Annex D)	Failure of communication peer Message corruption Message delay Message loss Unintended message repetition	Previous + Resequencing Insertion of message	Previous + Masquerading

NOTE 1 Higher DC can be claimed based on analysis. Likewise, lower coverage would result if the dominant failure mode is not listed.
NOTE 2 Transient faults are considered when shown to be relevant due, for instance, to the technology used.
NOTE 3 Failure modes for Processing Units can be adjusted to recognize a.c. fault models, such as transition faults (slow to rise and slow to fall nodes at application frequency) and path delays. Faults of this type are expected to be more prevalent with smaller process geometries. Usually tests for these types of faults are done at start-up, or power-down, or both, due to their intrusive nature and their ability to detect failures early with margin tests. Since they are hard to quantify, these failure modes are generally not included in failure rate calculations.
NOTE 4 If properly exercised, methods derived from stuck-at simulations (*e.g.* N-detect testing), but executed at application conditions, are known to be effective for d.c. fault and transition models as well.

[a] "Stuck-at": is a fault category that can be described with continuous "0" or "1" or "on" at the pins of an element. It is valid only for elements which have element level pin interfaces.
[b] "d.c. fault model" ("direct current fault model") includes the following failure modes: stuck-at faults, stuck-open, open or high impedance outputs, as well as short circuits between signal lines. It is not intended here to require an exhaustive analysis, for example to require the exhaustive analysis of bridging faults that can affect any theoretical combination of any signal inside a microcontroller or in a complex PCB. The analysis focuses on main signals or on very highly coupled interconnections identified with a layout level analysis.
[c] "soft error model": soft errors (*e.g.* bit flips) are the results of transient faults caused by alpha particles from package decay, neutrons, *etc.* These transient faults are also referred as Single Event Upset (SEU) and Single Event Transient (SET).

APPENDIX 2

Steering Column Lock System FMECA

Element	Failure Modes	Potential Causes	Local effects	Upper-Level Effect	Safety Level	System Safety Mechanisms (SSM)	Upper-Level Effect with SSM
ESCL-F1: Lock steering column	Spurious Lock ***ESCL-F1-FM1***	***µC-F1-FM1*** ***CB-F1-FM2*** ***MDB-F1-FM2*** ***MDB-F4-FM1*** ***MDB-F5-FM1*** ***MDB-F5-FM2*** ***SB-F-FM1*** ***SB-F-FM2***	Erroneous lock command	Steering column locked while driving ***SG1 Violated***	ASIL D	**SSM1:** Vehicle in motion **SSM2:** Switched power supply	No effect *
	ESCL-F1 Lost (No lock) ***ESCL-F1-FM2***	***CB-F1-FM1*** ***MDB-F1-FM1***	No lock command is possible	Parked vehicle with steering column unlocked		NA	
	ESCL-F1 stuck-at ***ESCL-F1-FM3***	***MDB-F3-FM2*** ***MDB-F4-FM2***	ESCL always performs lock command	Steering column remains locked => vehicle starts with locked column ***SG2 Violated***	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
ESCL-F2: Unlock steering column	Spurious Un-lock ***ESCL-F2-FM1***	***CB-F2-FM2*** ***MDB-F2-FM2***	Erroneous unlock command	Parked vehicle with steering column unlocked		NA	
	ESCL-F2 Lost No unlock) ***ESCL-F2-FM2***	***µC-F1-FM2*** ***CB-F2-FM1***	No unlock com-mand is possible	Steering column remains locked ➔ vehicle starts with locked column ***SG2 Violated***	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
	ESCL-F2 stuck-at ***ESCL-F2-FM3***	***MDB-F2-FM1***	ESCL always performs unlock command	Parked vehicle with steering column unlocked		NA	
BC-F1: Transmit Lock Com-mand from driver's interfaces to ESCL	Loss of BC-F1 ***BC-F1-FM1***	Out of our scope	No Lock Com-mand transmit to the ESCL	Parked vehicle with steering column unlocked		NA	
	Unintended ***BC-F1*** ***BC-F1-FM2***	Out of our scope	Unintended Lock Command trans-mits to the ESCL	Steering column locked while driving ***SG1 Violated***	ASIL D	**SSM1:** Vehicle in motion **SSM2:** Switched power supply **SSM4:** Plausibility check in the ESCL	No effects*
BC-F2: Transmit Unlock Command from driver's interfaces to ESCL	Loss of BC-F2 ***BC-F2-FM1***	Out of our scope	BC cannot transmit Unlock Command to the ESCL	Steering column remains locked ➔ vehicle starts with locked column ***SG2 Violated***	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effects *
	Unintended BC-F2 ***BC-F2-FM2***	Out of our scope	Unintended Un-lock Command transmits to the ESCL	Parked vehicle with steering column unlocked		NA	
BA-F: Transmit "vehicle in motion" to the ESCL	Loss of BA-F ***BA-F-FM1***	Out of our scope	BA-F does not provide "vehicle in motion" to the ESCL	Loss of ESCL-F1 and ESCL-F2	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
	Unintended permanent BA-F ***BA-F-FM2***	Out of our scope	BA-F provides "vehicle in mo-tion" to the ESCL permanently	Loss of the safety line "vehicle in motion" protection			
PS-F: Supply a switched electrical power to ESCL	Loss of PS-F ***PS-F-FM1***	Out of our scope	PS-F does not supply electrically ESCL	Loss of ESCL-F1 and ESCL-F2	ASIL A	**SSM3:** Monitoring of motor position should be implemented	No effect *
	Unintended permanent PS-F ***PS-F-FM2***	Out of our scope	PS-F supply permanently the ESCL electrically	Loss of a switched power supply protec-tion			

* assuming a perfect coverage of safety mechanisms

APPENDIX 3

Electronic Steering Column Lock (ESCL) Product FMECA

Element	Failure Modes	Potential Causes	Local Effects	Upper-Level Effect	Safety Level	Product Safety Mechanisms (PSM)	Product Effect with PSM
µC-F1: Control the state of the MDB	Erroneous assignment of outputs of the micro-controller ***µC-F1-FM1***	RAM, Flash, ROM Corruption, Oscillator, SW defect...	Spurious activation of the MDB locking state.	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM1:** Watchdog (HW), **PSM2:** 2 different SW modules should be implemented to control the µC-F1 (redundancy)	Safe State (ESCL shutdown)
	Loss of µC-F1 ***µC-F1-FM2***	RAM, Flash, ROM Corruption, Oscillator, SW defect...	The MDB state cannot be changed	ESCL-F2 Lost (No unlock) ***ESCL-F2-FM2***	ASIL A	**PSM3:** The µC should send periodically its status to the BCM	Safe state (ESCL shutdown)
CB-F1: Transmit Lock requests from BC to the µC	Loss of CB-F1 ***CB-F1-FM1***	LIN saturated, LIN opened...	No Lock Command transmit to the µC	ESCL-F1 Lost (no lock) ***ESCL-F1-FM2***		NA	
	Unintended CB-F1 ***CB -F1-FM2***	LIN saturated/corruption...	Unintended Lock Command transmits to the µC	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM4**: Plausibility check by µC	Safe state (ESCL shutdown)
CB-F2: Transmit Unlock requests from BC to the µC	Loss of CB -F2 ***CB -F2-FM1***	LIN saturated, LIN opened...	CB cannot transmit Unlock Command to the µC	ESCL-F2 Lost (No unlock) ***ESCL-F2-FM2***	ASIL A	**PSM5**: µC should verify if CB-F2 is alive	Safe state (ESCL shutdown)
	Unintended CB-F2 ***CB -F2-FM2***	LIN saturated/corruption...	Unintended Unlock Command transmits to the µC	Spurious Unlock ***ESCL-F2-FM1***		NA	
MDB-F1: The MDB locks the motor	Loss ***MDB-F1-FM1***	MDB command opened...	the MDB cannot lock the motor	ESCL-F1 Lost (No lock) ***ESCL-F1-FM2***		NA	
	Unintended ***MDB-F1-FM2***	Internal closed circuit...	Unintended lock of the motor by the MDB	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM6**: Plausibility check by µC	Safe state (ESCL shutdown)
MDB-F2: The MDB unlocks the motor	Loss ***MDB-F2-FM1***	MDB command opened...	the MDB cannot unlock the motor	ESCL-F2 stuck-at ***ESCL-F2-FM3***	ASIL A	**PSM7**: µC should verify if MDB-F2 is alive	Safe state (ESCL shutdown)
	Unintended ***MDB-F2-FM2***	Internal closed circuit...	Unintended unlock of the motor by the MDB	Spurious Unlock ***ESCL-F2-FM1***		NA	
MDB-F3: The MDB brakes the motor	Loss ***MDB-F3-FM1***	MDB command opened...	Deteriorate the motor	Not safety related		NA	
	Unintended ***MDB-F3-FM2***	Internal closed circuit...	Unintended brake of the motor by the MDB during unlocking	ESCL-F1 stuck-at ***ESCL-F1-FM3***	ASIL A	**PSM8**: Plausibility check by µC	Safe state (ESCL shutdown)
MDB-F4: The MDB unsupplies the motor	Loss (motor always supplied) ***MDB-F4-FM1***	MDB command closed (short circuit)	Unintended lock of the motor	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM9**: Plausibility check by µC	Safe state (ESCL shutdown)
	Unintended ***MDB-F4-FM2***	MDB command opened...	Unintended un-supply of the motor by the MDB	ESCL-F1 stuck-at ***ESCL-F1-FM3***	ASIL A	**PSM10**: µC should verify if MDB-F2 is alive	Safe state (ESCL shutdown)
MDB-F5: The MDB send the his status to the µC	Erroneous ***MDB-F5-FM1***	Drift / value coding	Erroneous MDB status sent to the µC	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM11:** µC must check the plausibility of sensor state input	Safe state (ESCL shutdown)
	Loss ***MDB-F5-FM2***	Open circuit	Erroneous MDB status sent to the µC	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM12:** µC must check the if SB alive	Safe state (ESCL shutdown)
SB-F: Monitor the position of the steering column	Loss of SB-F ***SB-F-FM1***	Drift / value coding	Erroneous sensor state sent to the µC	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM13:** µC must check the if SB alive	Safe state (ESCL shutdown)
	Erroneous SB-F ***SB-F-FM2***	Open circuit	Erroneous sensor state sent to the µC	Spurious lock ***ESCL-F1-FM1***	ASIL D	**PSM14:** µC must check the plausibility of sensor state input	Safe state (ESCL shutdown)

* assuming a perfect coverage of safety mechanisms

PUBLICATIONS

Pintard, L., Fabre, J.-C., Kanoun, K., Leeman, M., Roy, M., 2013. Fault Injection in the Automotive Standard ISO 26262: An Initial Approach, in: European Workshop on Dependable Computing 2013. pp. 126–133. doi:10.1007/978-3-642-38789-0_11

Pintard, L., Fabre, J.-C., Kanoun, K., Leeman, M., Roy, M., 2014. Fault Injection and Automotive Development, in: Embedded Real Time System and Software ERTS² 2014.

Pintard, L., Fabre, J.-C., Leeman, M., Kanoun, K., Roy, M., 2014. From Safety Analyses to Experimental Validation of Automotive Embedded Systems, in: Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on. pp. 125–134. doi:10.1109/PRDC.2014.23

Pintard, L., Leeman, M., Ymlahi-Ouazzani, A., Fabre, J.-C., Kanoun, K., Roy, M., 2015. Using Fault Injection to Verify an AUTOSAR Application According to the ISO 26262. SAE Technical Paper. http://papers.sae.org/2015-01-0272/

REFERENCES

Aidemark, J., Vinter, J., Folkesson, P., & Karlsson, J. (2001). GOOFI: generic object-oriented fault injection tool. *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, (pp. 83-88). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=941394

Albinet, A., Arlat, J., & Fabre, J.-C. (2004). Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04).* IEEE.

Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., . . . Powell, D. (1990). Fault injection for dependability validation: A methodology and some applications. *Software Engineering, IEEE Transactions on, 16*(2), 166-182.

Arlat, J., Costes, A., Crouzet, Y., Laprie, J.-C., & Powell, D. (1993). Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers, 42*(8), 913-923.

Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., & Leber, G. H. (2003). Comparison of physical and software-implemented fault injection techniques. *Computers, IEEE Transactions on, 52*(9), 1115-1133. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1228509

Aurum, A., Petersson, H., & Wohlin, C. (2002). State-of-the-Art: Software Inspections after 25 Years. *Software Testing, Verification and Reliability, 12*(3), 133-154.

AUTOSAR. (2015). *AUTOSAR Development Cooperation*. Retrieved from http://www.autosar.org

AUTOSAR_SWS_OS. (2014). *Specification of Operating System AUTOSAR Release 4.2.1.*

AUTOSAR-WDGM. (2014). *AUTOSAR Specification of Watchdog Manager AUTOSAR Release 4.2.1.*

Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on, 1*(1), 11-33.

Ayatolahi, F., Sangchoolie, B., Johansson, R., & Karlsson, J. (2013). A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution. In F. Bitsch, J. Guiochet, & M. Kaâniche (Eds.), *Computer Safety, Reliability, and Security* (Vol. 8153, pp. 265-276). Springer Berlin Heidelberg.

Barbosa, R., Karlsson, J., Madeira, H., & Vieira, M. (2012). Fault injection. In *Resilience Assessment and Evaluation of Computing Systems* (pp. 263-281). Springer.

Barbosa, R., Silva, N., & Cunha, J. M. (2013). csXception: First Steps to Provide Fault Injection. In M. Vieira, & J. Cunha (Eds.), *Dependable Computing, European Workshop on Dependable Computing* (Vol. 7869, pp. 202-205). Springer Berlin Heidelberg.

Barlow, R., & Lambert, H. (1975). Introduction to fault tree analysis. (R. (. Barlow, Ed.) *Conference on reliability and fault tree analysis*, pp. 7-35.

BARR Group. (2014). *Killer Apps: Embedded Software's Greatest Hit Jobs.* Retrieved 1 1, 2015, from http://www.barrgroup.com/killer-apps/

Benso A., D. C. (2011). The Art of Fault Injection. *SRAIT, 13*(4), 9-18.

Benso, A., Di Carlo, S., Di Natale, G., Prinetto, P., Solcia, I., & Tagliaferri, L. (2003). FAUST: fault-injection script-based tool. *9th IEEE On-Line Testing Symposium, IOLTS 2003.* Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1214386

Bidokhti, N. (2009). FMEA Is Not Enough. *Reliability and Maintainability Symposium, 2009. RAMS 2009. Annual* (pp. 333-337). IEEE. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4914698

Blin, A., Laarouchi, Y., & Quéré, P. (2014). Fault-Injection using Virtualization for Critical Software Validation in Automotive. *Embedded Real Time Software and Systems Congress (ERTS2), Toulouse (France)*, (p. 8). Retrieved from http://www.erts2014.org/

Bouti, A., & Kadi, D. A. (1994). A state-of-the-art review of FMEA/FMECA. *International Journal of reliability, quality and safety engineering, 1*(04), 515-543.

Carreira, J., Madeira, H., & Silva, J. (1998). Xception: A technique for the experimental evaluation of dependability in modern computers. *Software Engineering, IEEE Transactions on, 24*(2), 125-136.

Cherfi, A., Leeman, M., & Rauzy, A. (2014). Calculation of ISO 26262 Architectural Metrics From Fault Trees. (p. 10). Dijon: 19e Congrès de Maitrise des Risques de Sûreté de Fonctionnement.

Cherfi, A., Rauzy, A., & Leeman, M. (2014). AltaRica 3 Based Models for ISO 26262 Automotive Safety Mechanisms. *Model-Based Safety and Assessment IMBSA* (pp. 123-136). Munchen: Springer.

Christmansson, J., & Chillarege, R. (1996). Generation of an error set that emulates software faults based on field data. *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, (pp. 304-313). Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=534615

Chu, H. N. (2011). Test and Evaluation of the Robustness of the Functional Layer of. Institut National Polytechnique de Toulouse.

Costa, P., Silva, J., & Madeira, H. (2009). Dependability Benchmarking Using Software Faults: How to Create Practical and Representative Faultloads. *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, (pp. 289-294). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5369145

DBench. (2004). *DBench, European Project on Dependability Benchmarking.* (K. Kanoun, H. Madeira, Y. Crouzet, M. D. Cin, F. Moreira, & J.-C. R. Garcia, Eds.) Retrieved from http://www.laas.fr/DBench

Dees, R. (2012). *An Introduction to the IEEE-ISTO 5001 Nexus Debug Standard.* AMT Publishing.

DeMillo, R., Guindi, D., McCracken, W., Offutt, A., & King, K. (1988). An extended overview of the Mothra software testing environment. *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, (pp. 142-151).

Department of the Army, T. 5.-6.-4. (2006). *Effects Criticality Analysis (FMECA) for Command, Control, Communications, Computer, Intelligence, Surveillance, and Reconnaissance (C4ISR) Facilities.* Army Corp of engineers Power reliability enhancement Program (PreP).

ECSS-Q-30-02B. (2008). *Space product assurance: Failure modes, effects (and criticality) analysis (FMEA/FMECA).* ESA Requirements and Standards Division. Retrieved from http://www.ecss.nl/forums/ecss/dispatch.cgi/home/showFile/100704/d20080806093054/No/ecss-q-30-02b-draft2(30April2008).pdf

Fürst, S. (2008). *AUTOSAR – An open standardized software architecture for the automotive industry.* Retrieved 1 2015, 1, from http://www.autosar.org/: http://www.autosar.org/fileadmin/files/events/2008-10-23-1st-autosar-open/03_AUTOSAR_Tutorial.pdf

Gaudel, M.-C. (1995). Testing can be formal, too. *TAPSOFT '95: Theory and Practice of Software Development. 915*, pp. 82-96. Lecture Notes in Computer Science.

Han, S., Shin, K., & Rosenberg, H. (1995). DOCTOR: an integrated software fault injection environment for distributed real-time systems. *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, (pp. 204-213). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=395831

Hsueh, M.-C., Tsai, T. K., & Iyer, R. K. (1997). Fault injection techniques and tools. *Computer, 30*(4), 75-82. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=585157

IEC 61508. (2010). Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems.

IEEE Std. (2006). IEEE Standard e Functional Verification Language. *(1647)*. IEEE Standard.

IFIP WG 10.4. (2015). Retrieved from IFIP Working Group on Dependable Computing and Fault Tolerance: http://www.dependability.org/wg10.4/

Islam, M. M., Sangchoolie, B., Ayatolahi, F., Skarin, D., Vinter, J., Törner, F., . . . Karlsson, J. (2013). Towards Benchmarking of Functional Safety in the Automotive Industry. In M. Vieira, & J. C. Cunha (Ed.), *14th European Workshop, EWDC 2013,. 7869*, pp. 111-125. Coimbra, Portugal: Lecture Notes in Computer Science.

Islam, M., Karunakaran, N., Haraldsson, J., Bernin, F., & Karlsson, J. (2014). Binary-Level Fault Injection for AUTOSAR Systems. *Dependable Computing Conference (EDCC), 2014 Tenth European*, (pp. 138-141). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6821098

ISO 26262. (2011). Road Vehicles - Functional Safety. International Organization for Standardization / Technical Committee 22 (ISO/TC 22). Retrieved from http://www.iso.org/iso/home/news_index/news_archive/news.htm?refid=Ref1499

Jenn, E., Arlat, J., Rimbn, M., Ohlsson, J., & Karlsson, J. (1994). Fault Injection into VHDL Models: The MEFISTO Tool. *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-*

Fourth International Symposium on (pp. 66-75). IEEE. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=315656

Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., & Karlsson, J. (1994). Fault Injection Into VHDL Models: A Fault Injection Tool And Some Preliminary Experimental Results. *Integrating Error Models with Fault Injection, 1994., Third Int. Workshop on*, (pp. 13-14). From http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=654393

Kaâniche, M., Romano, L., Kalbarczyk, Z., Iyer, R., & Karcich, R. (1998). A Hierarchical and Approach for Dependability and Analysis and of a Commercial and Cache-Based RAID and Storage Architecture. *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on* (pp. 6-15). IEEE. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=689450

Kanawati, G., Kanawati, N., & Abraham, J. (1995). FERRARI: a flexible software-based fault and error injection system. *IEEE, 44*(2), 248-260. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=364536

Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber, G., & Reisinger, J. (1998). Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. *DEPENDABLE COMPUTING AND FAULT TOLERANT SYSTEMS, 10*, 267-288.

Karlsson, J., Liden, P., Dahlgren, P., Johansson, R., & Gunneflo, U. (1994). Using heavy-ion radiation to validate fault-handling mechanisms. *Micro, IEEE, 14*(1), 8-23.

Karunakaran, N. M. (2013). Binary-Level Fault Injection (BLFI) for AUTOSAR-based Systems. Department of Computer Science and Engineering.

Koopman, P. (2014). *A Case Study of Toyota Unintended Acceleration and Software Safety.* Retrieved from Better Embedded System SW: http://betterembsw.blogspot.fr/2014/09/a-case-study-of-toyota-unintended.html

Koopman, P. (2014). *A Case Study of Toyota Unintended Acceleration and Software Safety.* From Better Embedded System SW: http://betterembsw.blogspot.fr/2014/09/a-case-study-of-toyota-unintended.html

Koopman, P., DeVale, K., & DeVale, J. (2008). Interface RobustnessTesting: Experiences and Lessons Learned From The BALLISTA Project. In K. Kanoun, & L. Spainhower (Eds.), *Dependability Benchmarking for Computer Systems.* (pp. 201-226). Copyright © 2008 IEEE Computer Society.

Lanigan, P., Narasimhan, P., & Fuhrman, T. (2010). Experiences with a CANoe-based fault injection framework for AUTOSAR. *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, (pp. 569-574). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5544419

Laprie, J.-C., Arlat, J., Blanquart, J.-P., Costes, A., Crouzet, Y., Deswarte, Y., . . . Thévenod, P. (1995). *Dependability Handbook - Le Guide de la Sûreté de Fonctionnement.* Cépaduès-Editions, Toulouse.

Lauterbach. (2015). Retrieved 01 01, 2015, from Lauterbach development tools: http://www.lauterbach.com

Leeke, M., & Jhumka, A. (2009). Beyond the golden run: evaluating the use of reference run models in fault injection analysis. *Proceedings of the 25th UK Performance Engineering Workshop*, (pp. 61-74). Retrieved from http://wrap.warwick.ac.uk/47538

Leeman, M. (2013). The deployment of ISO26262 in Valeo. Paris: SIA.

L'Hostis, S. (2013). Architectural metrics calculation – an efficient approach. *Sécurité fonctionnelle électronique automobile : ISO 26262 : où en sommes-nous?* Paris: SIA.

Lu, C. (2009a). *Robustesse du logiciel embarqué multicouche par une approche réflexive: application à l'automobile.* Ph.D. dissertation, Institut National Polytechnique de Toulouse (INPT).

Lu, C., Fabre, J.-C., & Kilijian, M.-O. (2009b). An approach for improving Fault-Tolerance in Automotive and Modular Embedded Software. *17th International Conference on Real-Time and Network Systems RTNS'2009, Paris, ECE, 26-27 October.* IEEE.

M2OS. (2014). *Method Sheets*. (M. S. M2OS: Management, Editor, & IMdR) Retrieved 01 01, 2015, from http://www.imdr.eu/upload/client/document_site/Fiches_pedago_8_20140610_EN.pdf

Madeira, H., Rela, M., Moreira, F., & Silva, J. G. (1994). RIFLE: A General Purpose Pin-level Fault Injector. *In Dependable Computing—EDCC-1. 852*, pp. 197-216. Springer Berlin Heidelberg.

Mariani, R., & Boschi, G. (2007). A systematic approach for Failure Modes and Effects Analysis of System-On-Chips. *0*, pp. 187-188. Los Alamitos, CA, USA: IEEE Computer Society.

Mariani, R., Boschi, G., & Colucci, F. (2007). Using an innovative SoC-level FMEA methodology to design in compliance with IEC 61508. *in Proceedings of the conference on Design, automation and test in Europe, DATE '07*, (pp. 492–497). (San Jose, CA, USA). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4211846

Mariani, R., Fuhrmann, P., & Vittorelli, B. (2006). Fault-robust microcontrollers for automotive applications. *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, (pp. 6 pp.-).

NASA. (2005). *Glossary - NASA Crew Exploration Vehicle, SOL NNT05AA01J, Attachment J-6*. Retrieved 09 18, 2014, from http://www.spaceref.com/news/viewsr.html?pid=15201

Natella, R. (2011). *Achieving Representative Faultloads in Software Fault Injection.* Ph.D. dissertation, Università Degli Studi di Napoli Federico II.

Natella, R., Cotroneo, D., Duraes, J., & Madeira, H. (2013). On Fault Representativeness of Software Fault Injection. *IEEE, 39*(1), 80-96. Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6122035

Nexus5001. (2015). Retrieved 01 01, 2015, from http://nexus5001.org/

Park, J.-D., Yi, C.-H., Kwon, K.-H., & Jeon, J. W. (2014). Method of fault injection for medical device based on ISO 26262. *Consumer Electronics (ISCE 2014), The 18th IEEE International Symposium on* (pp. 1-2). IEEE.

Piper, T., Winter, S., Manns, P., & Suri, N. (2012). Instrumenting AUTOSAR for dependability assessment: A guidance framework. *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, (pp. 1-12). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6263913

Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., & Törner, F. (2013). Improving Fault Injection in Automotive Model Based Development using Fault Bypass Modeling. *in 2nd Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES'13)*, (pp. 2577-2591). Koblenz, Germany.

RTCA & EUROCAE. (2011). DO-178C/ED-12C. *Software Considerations in Airborne Systems and Equipment Certification*. Retrieved from http://www.rtca.org/store_product.asp?prodid=803

SAE International. (2010). ARP 4754. *Certification Considerations for Highly-Integrated Or Complex Aircraft*.

Salkham, A., Pecchia, A., & Silva, N. (2013). Design of a CDD-Based Fault Injection Framework for AUTOSAR Systems. *Proceedings of Workshop SASSUR (Next Generation of System Assurance Approaches for Safety-Critical Systems) of the 32nd International Conference on Computer Safety, Reliability and Security.* Retrieved from http://hal.archives-ouvertes.fr/hal-00848500/

SaRS: Safety and Reliability Society. (2011). *Chapter 38 Markov Modeling.* Applied R&M Manual for Defence Systems Part C - R&M Related Techniques.

SaRS: Safety and Reliability Society. (2011). *Chapter 30 Reliability Block Diagrams.* Applied R&M Manual for Defence Systems Part C - R&M Related Techniques .

Shepardson, D. (2015, 02 13). *U.S. auto recalls hit 63.95 million in 2014*. Retrieved from Detroit News: http://www.detroitnews.com/story/business/autos/2015/02/12/auto-recalls/23307005/

Silva, N., Barbosa, R., Cunha, J. C., & Vieira, M. (2013). A view on the past and future of fault injection. *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, (pp. 1-2).

Simulink. (2015). Retrieved 01 01, 2015, from http://www.mathworks.com/products/simulink/

Skarin, D., Barbosa, R., & Karlsson, J. (2010, June). GOOFI-2: A tool for experimental dependability assessment. *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, (pp. 557-562).

Stateflow. (2015). Retrieved 01 01, 2015, from http://www.mathworks.com/products/stateflow/

Statemate. (2015). Retrieved 01 01, 2015, from http://www-03.ibm.com/software/products/en/ratistat

STMicroelectronic. (2013). SPC56EL70 32 bit PowerArchitecture® microcontroller for automotive SIL3/ASIL D chassis and safety applications.

Strong, M. (2015, February 2). *Honda Confirms New Takata Fatality*. Retrieved from The Detroit Bureau: http://www.thedetroitbureau.com/2015/02/honda-confirms-new-takata-fatality/

Svenningsson, R. (2011, december). Model-Implemented Fault Injection for Robustness Assessment. *Model-Implemented Fault Injection for Robustness Assessment(2011:16)*, xi, 39. KTH Royal Institute of Technology.

Svenningsson, R., Eriksson, H., Vinter, J., & Törngren, M. (2010). Model-Implemented Fault Injection for Hardware Fault Simulation. *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on*, (pp. 31-36). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5772248

Svenningsson, R., Vinter, J., Eriksson, H., & Törngren, M. (2010). MODIFI: A MODel-Implemented Fault Injection Tool. In E. Schoitsch (Ed.), *Computer Safety, Reliability, and Security (SAFECOMP 2010). 6351*, pp. 210–222. © Springer-Verlag Berlin Heidelberg 2010. Retrieved from http://dx.doi.org/10.1007/978-3-642-15651-9_16

Trawczynski, D., Sosnowski, J., & Gawkowski, P. (2008). Analyzing fault susceptibility of ABS microcontroller. *Computer Safety, Reliability, and Security* (pp. 360-372). Springer Berlin Heidelberg.

Vinter, J., Bromander, L., Raistrick, P., & Edler, H. (2007). FISCADE - A Fault Injection Tool for SCADE Models. *Automotive Electronics, 2007 3rd Institution of Engineering and Technology Conference on*, (pp. 1-9). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4383624

Xi, C. (2008). *Requirements and concepts for future automotive electronic architectures from the view of integrated safety.* Ph.D. dissertation, Universität Karlsruhe (TH).

Yogitech. (2015). Retrieved from http://www.yogitech.com/en

Yuste, P., de Andrès, D., Lemus, L., Serrano, J. J., & Gil, P. J. (2003). INERTE: Integrated NExus-Based Real-Time Fault Injection Tool for Embedded Systems. *in Dependable Systems and Networks, DSN 2003. Proceedings. 2003 International Conference on*, (p. 669). Retrieved from http://www.computer.org/csdl/proceedings/dsn/2003/1952/00/19520669.pdf?origin=publication_detail

Ziade, H., Ayoubi, R., & Velazco, R. (2004, July). A Survey on Fault Injection Techniques. *The International Arab Journal of Information Technology, 1*(2), 171-186.