Using Fault Injection to Verify an AUTOSAR Application According to the ISO 26262

Ludovic Pintard, Michel Leeman, Abdelillah Ymlahi-Ouazzani, Jean-Charles Fabre, Karama Kanoun, Matthieu Roy

Using Fault Injection to Verify an AUTOSAR Application According to the ISO 26262

Ludovic Pintard, Michel Leeman, and Abdelillah Ymlahi-Ouazzani
VALEO
Jean-Charles Fabre, Karama Kanoun, and Matthieu Roy
LAAS-CNRS

CITATION: Pintard, L., Leeman, M., Ymlahi-Ouazzani, A., Fabre, J. et al., "Using Fault Injection to Verify an AUTOSAR Application According to the ISO 26262," SAE Technical Paper 2015-01-0272, 2015, doi:10.4271/2015-01-0272.
http://papers.sae.org/2015-01-0272

Abstract
The complexity and the criticality of automotive electronic embedded systems are steadily increasing today, and that is particularly the case for software development. The new ISO 26262 standard for functional safety is one of the answers to these challenges. The ISO 26262 defines requirements on the development process in order to ensure the safety. Among these requirements, fault injection (FI) is introduced as a dedicated technique to assess the effectiveness of safety mechanisms and demonstrate the correct implementation of the safety requirements.
Our work aims at developing an approach that will help integrate FI in the whole development process in a continuous way, from system requirements to the verification and validation phase. This leads us to explore the benefits of safety analyses (Failure Mode Effects and Criticality Analysis (FMECA), Fault Tree Analysis (FTA), Critical Path Analysis (CPA) or Freedom From Interference (FFI) Analysis, etc.) for the definition of the test plan, defining efficient FI tests cases.
The paper discusses the objectives and role of FI in the Verification and Validation process. It also illustrates how to apply this methodology on a platform based on AUTOSAR 4.X that integrates a trusted Front-Light Manager Application (Automotive Safety Integrity Level - ASIL B) and a non-trusted (Quality Management - QM) application. This proposed architecture allows ensuring the safety requirements with dedicated safety mechanisms and also FFI using both temporal and spatial partitioning. Finally, the results of FI test cases obtained on a mock-up running the Front-Light Manager Application, developed at Valeo GEEDS are presented.

Introduction
To deal with the criticality and the increasing complexity of electrical and electronic automotive systems, the automotive industry has introduced the ISO 26262 standard [1] for functional safety in road vehicle. This standard has pushed recommendations towards different methods and techniques that should be adopted in a development process to assure that "no unreasonable risk is due to hazards caused by malfunctioning behavior of electrical and electronic systems".
Fault injection (FI) is one of the dedicated techniques to achieve the verification of safety requirements. FI is now a method highly recommended in the ISO 26262 (10 requirements are about the use of FI at different steps of the V-cycle), particularly, in Part 6 on software developments. FI allows the verification of the effectiveness of safety mechanisms, as these cannot be activated using other test methods.
Section 1 introduces the notions of safety analyses and fault injection. Section 2 aims at defining the objectives of fault injection in the verification and validation process based on the results of the safety analyses. Section 3 describes the software architecture of the case study: the Front-Light Manager. Section 4 applies the recommendations of the section 2 on the case study. Finally, the last section concludes the paper.

1. State of the Art

ISO 26262 Standard for Functional Safety

Safety Development Process
In a previous work [2], the integration of FI in the early phases of the development process of automotive critical systems, in the context of the ISO 26262 standard, has been investigated. The synthesis of FI Analyses into Failure Mode Effects and Criticality Analysis -- FMECA spreadsheets has been demonstrated. The process also helps to capture the failure propagation paths between the levels of architecture. The interest of these analyses is twofold: i) they identify the nature and location of the safety mechanisms that must be integrated in the system, ii) they guide FI experiments during post-implementation phase.
The critical paths identified in safety analyses improve significantly the traceability of safety requirements.

Ensuring Freedom From Interferences (FFI)
Nowadays, the processing capabilities of the microcontrollers used for automotive systems allow designing more complex products. Particularly, the software design takes advantage of these resources, by proposing to integrate 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 criticality levels, i.e., different Automotive Safety Integrity Level (ASIL). According to the ISO 26262, all the modules on the 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, according to the ISO 26262, to be developed with unnecessary efforts. Indeed, higher ASIL modules require applying more techniques and methods.
However, the ISO 26262 allows integrations of modules with different ASILs, but imposes to prove that the Freedom From Interferences (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". In a more practical way, the integration of a Quality Management (QM) or lower ASIL module is allowed if and only if it could be proved that it does not interfere directly or indirectly with the behavior of any higher ASIL 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 (scheduling, task pre-emption…), iv) shared-memory access and v) shared hardware peripherals.

Fault Injection in Verification & Validation Activities

Fault Injection Techniques

Fault injection [3] is a mature technology that has been successfully applied using several techniques on different targets [4, 5, 6].
Fault injection techniques have been developed for a long time, first, as a mean to validate safety mechanisms (error detection and recovery). The fault model is an input for the design of fault tolerant systems; the development of error detection and recovery mechanisms should efficiently perform error processing. The efficiency is evaluated using measures like Error Detection Coverage (EDC), as a conditional probability. When a fault belonging to the fault model is injected, the corresponding error detection mechanisms should be activated and trigger recovery mechanisms if any. Secondly, fault injection can also be used to evaluate the robustness of a component (hardware – HW – or software – SW) with respect to some fault assumptions. The idea here is often to evaluate the efficiency of built-in error detection mechanisms to prevent failure modes violating safety requirements.
Moreover, the introduction of FI, in the ISO 26262, renewed the interest of this topic in the automotive industry [7, 8, 9].

FI completes other testing techniques by enabling to verify the global reaction of an application against systematic or random faults from shared components. Moreover, both random (HW aging, single event upset) faults and systematic (coding/implementation) faults are assessed at system and product levels. At software level, FI is only applied to verify systematic faults' absence in the considered application.

FARM

The main elements that characterize FI are: 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 the readouts. These elements form the basis of the **FARM** [10] model to perform FI.
The **FARM** model characterizes in an effective way the fault injection environment, and it is used in this study as a reference in the definition of FI experiments.
It is worth to mention that the measures to be evaluated have to be defined first, since they guide the whole FI process. However, their values are evaluated in the last step, by processing information provided by the readouts. The fault model, the system activation and the readouts have to be defined in such a way that they enable obtaining the measures in a meaningful and efficient way.
FARM is essential in the paper, as all the FI experiments have been defined following this concept.

2. Fault Injection for the Verification & Validation

The importance of the safety analyses in the identification of the critical elements and of the critical propagation paths, have been shown. Moreover, several detection and mitigation means are identified and must be implemented in order to ensure the safety requirements at each level of architecture.
This section will show how safety analyses help in the definition of FI tests. Particularly, by giving dedicated measures on the coverage of the safety mechanisms and the correct mitigation of critical paths, fault injection enables to verify the implementation of safety requirements and of FFI.

Concerning verification and validation activities, fault injection is required by the ISO 26262, in parts 4, 5 and 6. This study focuses on the requirements of the part 4, i.e., system level, and part 6, i.e., software level.

ISO 26262 Requirements on FI

SW Unit Testing & SW Integration Testing

At these two levels, fault injection *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) robustness; and

EXAMPLE: The absence of inaccessible software, the effectiveness of error detection and error handling mechanisms.

f) *sufficient resources to support their functionality.*

At this level fault injection, *i.e. that includes the injection of arbitrary faults (by corrupting values of variables, by introducing code mutation, or by corrupting values of Central Processing Unit (CPU) registers)* is at least recommended at all levels and also highly recommended for ASIL D and specifically for ASIL C at SW Integration testing.
Moreover, the ISO 26262 requires to perform "Requirements-based test". This implies that the safety mechanisms identified to ensure the safety requirements of any ASILs must be tested. The definition of these tests leads to use fault injection techniques. Hence, FI experiments should be performed for all ASILs. It enables to demonstrate confidence in the absence of unintended functionality or in the robustness of the target (Software unit or SW architecture).

To conclude at system level, fault injection must be done for all ASIL levels in order to verify the propagation of potential causes identified in the safety analyses, verify the error detection coverage of the implemented safety mechanisms. The injection of arbitrary faults is only required for ASIL D, and also ASIL C only for the SW integration tests. Only highest levels are required as the test efforts increase significantly as fault should be injected intensively in the software, with random fault injection.

System Testing

In part 4 of the ISO 26262, fault injection is required at HW/SW integration level and at product and system level (according to the chosen naming).
At these levels, ISO 26262 requires fault injection to:

- Demonstrate the effectiveness of the safety mechanisms' diagnostic coverage.
- Demonstrate the correct implementation of the safety requirements.

Similarly to Part 6, fault injection is highly recommended at least down to ASIL C, and specifically at ASIL B to demonstrate the correct implementation of safety requirements at HW/SW integration level. Moreover, Requirements-based tests are also highly recommended in order to verify functional and non-functional requirements. Fault injection technique must be used in order to validate the safety requirements at all ASIL.

Objectives and Definition of Fault Injection Tests

In this section, the applicability of the FARM method, defined in section 1, in the identification of the objectives of the fault injection tests and the planning of the tests is discussed.

Definition of the Experimentation Target

The safety analyses help to identify the most critical elements and the most critical propagation paths of the architecture thanks to the traceability between levels of the ASIL. These critical elements and paths should be particularly verified, in order to prove the causes of safety requirement violation have been mitigated. The first

experimentation targets are the one with the highest ASIL. The targets are, then, chosen following the possible target with decreasing criticality.
Nonetheless, the chosen targets usually impose the implementation of, at least, one safety mechanism. Then, it is also required to demonstrate efficiency against the considered fault model.

Measures to be Assessed

Fault injection has the following two main objectives. On the one hand, the verification that the safety requirements are not violated. The considered fault model from the safety analyses does not propagate through critical paths, and violate safety requirements. This can be quantified by identifying the failure modes distribution, in which the failure modes are associated with safety requirements. The failure modes distribution could be represented using "pie chart", "bar graphs" or "histograms". It is possible to evaluate the rate of faults that lead to violate a safety requirement.
On the other hand, FI addresses the verification of the effectiveness of the safety mechanisms. The considered fault model from the Safety Analyses is well handled by the safety mechanisms. Here, the aim is to assess the Error Detection Coverage and/or the Error Recovery Coverage (ERC) of the Safety Mechanism according to the fault model identify in the analyses. Generally, these results are represented as "pie chart", in order to illustrate the difference between the covered faults and the lack of coverage.
However, these measures only quantify the effectiveness of the safety mechanisms and the fault tolerance of proposed architecture in order to prevent the occurrence of Undesired Events. Then, there are also some qualitative measures that can be obtained with fault injection.

Hence, it could help to identify a lack of coverage of a safety mechanism, particularly to the "Diagnostic Coverage" or "coverage factor" that have been claimed and used for quantitative safety analyses. The main difference is the considered fault model, because it implies to verify the entire fault model that the safety mechanism claims to handle. Besides, a lack of coverage could lead to:

- Re-implement the safety mechanism.
- Add another mechanism in order to mitigate specific faults or modify the design.

Finally, when assessing the failure distribution, it could reveal failure modes that have not been identified during the Pre-implementation phase. There are two main reasons. On the one hand, the Safety analyses have been badly executed because the fault propagation of the fault model is incorrect or the failure modes are non-trivial due to the complexity of the design. On the other hand, the specification of functional requirements and safety requirements are incorrect. Indeed, these faults could lead to unexpected behavior if there is a lack of requirements or too many that interfere with each other.
These qualitative results aim at validating the assumptions and choices done in safety analyses.

Faults to be Injected

The fault model can be extracted from the safety analyses of potential causes of failures, identified for each critical elements or path. Two FI strategies are possible:

- Verify that an identified failure mode, e.g., in a FMECA row, is handled correctly by the implemented safety mechanisms. Hence, by injecting representative causes of failures leading to the considered failure mode, the identified safety requirements will be solicited, and will be easily tested.
  Let us consider, there are "n" potential causes that have been identified in a safety analysis and enable to activate the "n" equivalence classes (EC) of the failure mode (FM) of the SW module. Injecting these causes enable to verify that the safety mechanism mitigates correctly the failure mode of the SW module or find lacks in the coverage.
- Here, the safety analyses lead to identify a set of potential causes. Experiments consist in injecting all possible causes to check that propagation paths identified with the safety analyses are valid. In the latter case, it must be verified that the propagation is well mitigated.

During this activity, the strategy is chosen independently from FI capabilities provided by the available FI tool. Here, the goal is only to define the set of fault that should be injected in order to obtain the desired measures.

Activation Model:

The Activation model is a set of data patterns that aims at exercising the injected fault. For a general purpose, a solution consists of using a representative environment of the target. It aims at evaluating the behavior of the system in presence of faults during the representative uses of the target. They could be chosen according to the frequency, the criticality, etc. of the target.
Nonetheless, the determination of the Activation set could be significantly improved using behavioral models of the dynamic of the target with its environment. The behavioral models developed during the design of the system could be very useful for this purpose. Indeed sequence diagrams, timing diagrams or use cases done to describe both functional and non-functional behavior, are of prime importance in the definition of representative fault injection tests. On this basis, it is possible to determine when to trigger the injection i) from the states of the target's components and ii) inputs from the environment.

Readouts:

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

To define these readouts, it is recommended using the safety analyses; particularly the following FMECA columns: the failure modes, the effects (local effects and effects on higher levels of architecture), and the column describing the safety mechanisms.
These columns describe the various effects of the propagation of the faults, and hence should be used to define the variables (physical or digital) that have to be observed and acquired on the target. Further details about these measurement points (state or events e.g., timestamps, log files of variables) 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.

Table 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)*	*No Observation (No effect but to be analyzed)*

Then, from the readouts, a logical expression, or safety property, of the target states can be established to detect when a safety requirement is violated.
The analysis of the readouts leads to assess the measures defined at the beginning of the process. However, the results could be ambiguous and must be analyzed. These experiments have been classified into four categories, see Table 1. Here, the experiments

have been done on a target implementing at least one safety mechanism that prevents the violation of a safety requirement. This table ignores faults that have not been activated and that usually fall into the "no observation" category.

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 lack of coverage of the implemented safety mechanism(s). Case γ and δ often mean a design or implementation problem since the safety mechanisms have not been activated. Defaults in the fault injection experiments can also be a reason for this last case: i) an error remain latent and has not been activated by test 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).

Nonetheless, in a conventional fault injection campaign, the output is often a pie chart composed of the previous Cases. Ideally, 100% of errors are detected and recovered; hence the non-infringement of the safety requirement is ensured. In reality, some errors are not detected by internal error detection mechanisms or not recovered. Upper level safety mechanisms should then prevent the violation of safety requirement.

To conclude, safety analyses check/recommend safety mechanisms to be put in place, and FI tests quantify their efficiency, namely detection and recovery coverage. As the safety mechanisms may prevent undesired events (UEs) to be reached, this will enable to verify the safety requirements or the FFI is ensured.
When a safety property is violated the conclusion is twofold: i) lack of coverage of a safety mechanism. The implementation of the mechanism should be analyzed in order to improve the coverage, if necessary, and ii) absence of a safety mechanism leading thus to a revision of the design.

3. Case Study Description: the Front-Light System

The approach is illustrated in this paper by a Front-Light system. This electronic system controls the two headlights of a car.
Authors are aware 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 approach can be done.
In order to contextualize the SW architecture, a first part is dedicated to the description of system level and product level. Indeed, the objectives of fault injection tests are required based on assumptions and analyses done at higher levels of architecture. It enables the traceability of the requirements and therefore the identification of efficient fault injection tests cases. Then, the considered targets for the tests are especially the SW architecture and the Watchdog Manager (WdgM).
This section aims to introduce the section 4 on the verification and the validation of this architecture.

Description of the System and the Product Architectural Level

The Front-Light system has two system functional requirements:

- The Front-Light system must light the road on the request of the car's driver
- The Front-Light system must set the dashboard light indicator

A Preliminary Hazard Analysis (PHA) identifies two UEs (see Table 2). There are two Safety-related UEs, UE01 and UE02 rated ASIL B.

Table 2. UEs from the PHA and their ASIL Level

UEs #	Description	ASIL
UE01	Loss of the Headlights	ASIL B
UE02	Headlights Blinking	ASIL B

Hence, two safety requirements, referred to as Safety Goal (SG), can be defined. An ASIL has been allocated to this SG according to a Preliminary Hazard Analysis (PHA).
SG1: The system shall not spuriously cut off both Headlights (ASIL B)
SG2: The system shall prevent the Headlights blinking (ASIL B)

It could be noted that quantitative analyses are considered out of the scope of the study.

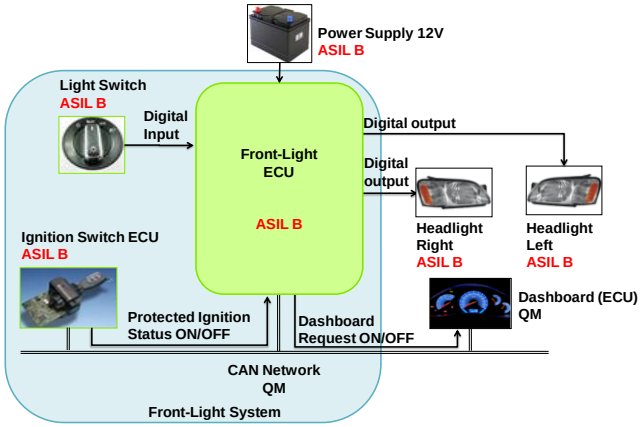

Figure 1. Architecture of the Front-Light System

Description of the System Architecture Level

The system architecture is defined in the Figure 1. The main product is the Front-Light Electronic Control Unit (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. Then the Front-Light ECU 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 3

Table 3. Description of the Functions of the Front-Light System. Focus on the Front-Light-ECU.

Product	Product Function Id #	Product Functional Requirements
Front-Light ECU	FL-ECU_F01	Front-Light ECU must send Dashboard State (ON/OFF) through 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 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

Then, a FMECA has been performed in order to identify the failure modes that may lead to violate the SGs (two failure modes have been found for the Front-Light-ECU, see Table 4). The FMECA also enables to allocate an ASIL to products and define System Safety Mechanisms.

Table 4. Failure Modes of the Front-Light-ECU that Lead to Violate a Safety Goal Identified in the FMECA

Product-UEs #	Product Function Id #	Failure mode description	ASIL
P-UE01	FL-ECU_F01	Loss of Headlights state	ASIL B
P-UE02		Unstable Headlights state	ASIL B

Information at the Product Level

At this, level, only the micro-controller that runs the software applications is considered, hence design and the solution chosen for the electronic card are not detailed.
The inputs and outputs links at product level are similar to the ones considered at system level. They do not add new pertinent failure modes. Particularly, the two P-UEs (see Table 4) defined could be considered also as two SW-UEs (SW-UE01 and SW-UE02).
However, the following failure modes of the micro controller should be considered: hardware failures and also the FFI between the applications running on the microcontroller.

Hardware Platform

The microcontroller Leopard SPC56EL70 [11] is based on PowerPC architecture and involves two identical processors (e200z4d cores) connected to a single, shared main memory. The microcontroller could be configured in lockstep or decoupled modes.
In lockstep mode, the two cores run the same instructions and then compare the results. It is required to ensure the safety of critical functions (e.g., ASIL C and D) by offering a tolerance on transient hardware faults. The decoupled mode enables to execute different instructions on each core, and therefore execute several tasks in parallel. The resources offer in this mode could be used in order to enhance the performance of the application and/or implement safety mechanisms.
This microcontroller is a safety requirements for a Safety Element out of Context (SEooC), see ISO 26262 chapter 10 [1], and meets ASIL D requirements.

Freedom From Interference (FFI) Analysis

The FFI may identify several causes of malfunction of the higher ASIL application, by analyzing the interference channels. The identified causes also require mitigation means – definition of safety mechanisms – in order to prevent the violation of a safety goal.

Without going into details, two applications with different criticality levels have been allocated on the microcontroller. There are respectively an ASIL B application and a QM application, which correspond respectively to functions, FL-ECU_F01 and FL-ECU_F02.
In a practical way, 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 of the ASIL B application.
4. **Shared Memory Interferences**: e.g., corruption of Critical data by the QM application through shared-memory (ROM, RAM, stack)

Each of these interferences can be defined as software UEs and they are respectively referenced as: SW-UE03 to SW-UE06. These UEs are all rated ASIL B as their occurrences could cause the violation of SG1 or SG2.

In order to provide an execution environment that allows the execution of software components without unintended interference, temporal and spatial partitioning for both computational and communication resources is required.

Spatial Partitioning: Spatial partitioning ensures that one software component cannot alter the code or private data of another software component. Spatial partitioning also prevents a software component from interfering with control of external devices (e. g., actuators) of other software components.
Temporal Partitioning: Temporal partitioning ensures that a software component cannot affect the ability of other software components to access shared resources, such as the common network or a shared CPU. This includes the temporal behavior of the services provided by resources (latency, jitter, duration of availability during a scheduled access).

Description of the SW Architecture

AUTOSAR: AUTomotive Open System Architecture

AUTOSAR [12] is a standard for automotive E/E software architectures developed by major OEMs and suppliers. It is a major aspect of the software development in the automotive industry.
AUTOSAR brings in the realization of an application-specific approach for automotive software development as opposed to an ECU-specific one. Hence, it provides a 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 (SWC), 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.
These layers are decomposed into five stacks (each stack is cross-layer): the Service stack, the Memory stack, the Communication stack, the Input/Output Hardware Abstraction stack, and the Complex Devices Drivers stack.

In order to implement the partitioning requirement on the FFI, the following AUTOSAR concept and modules are used.

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 [13], 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.

MMU/MPU: The basic memory protection requirement that shall be fulfilled by the OS is to protect data, code and stack section of OS-Application. In 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, it could be also used to protect private data and stack within the same OS-application if necessary.
The memory protection requires hardware support in the microcontroller of a MPU and/or MMU.

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

Software Architecture of the Front-Light Manager

The SW architecture is quite simple. All is controlled by an AUTOSAR-OS developed according the highest ASIL on the microcontroller, i.e., ASIL B.

As defined in the FFI Analysis, there are two OS-Applications. The trusted one manages the safety critical functionalities (ASIL B). It encompasses: four SWC (Switch Event, Light Request, Front-Light Manager, and Headlight), a trusted RTE, Systems Services (BSWM, WdgM), IO HW Abstraction Stack, IOC, and E2E (to unwrap the protection of the switch light Status). This OS application runs the critical runnables of the SWCs. They aim at gathering the information of the light switch and the ignition switch and to process the states of the Dashboard indicator and the Headlights.

Besides, the non-trusted OS-Application is composed of two SWCs (*ComStackDemoApp, and another application: SystackDemoApp)*, a Non-trusted RTE, and the communication Stack for CAN transmissions and receptions.

The E2E protection is an important point in the design. The signal from the ignition switch is ASIL B, but the signal is process by modules of the QM OS-Application. The E2E protection prevents from the data corruptions by the QM application, and ensure the requirements of ASIL B, all along the critical path.

A software FMECA has been processed on the architecture described in Figure 2, to detail the fault propagation paths on the software architecture. An example extracted from the complete spreadsheet is given in the Table 5. It could be noted that the complete table is composed of 116 lines or Failure modes. 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: out of range, valid error, or data loss), function call errors (function not called, function call with wrong arguments).

Safety mechanisms have been identified in order to handle the failure modes of the software modules. Three aliveness supervisions of the WdgM are configured to ensure that the critical SWCs are still executed. The aliveness of ComStackDemoApp and the Can stack are also monitored to prevent interference on the Ignition Switch status provided to the Light Request. Control flow and deadline supervisions are implemented to monitor the execution of the ASIL B SWCs.

In this application, valid errors provided by several modules in the critical path may not be detected and lead to a safety requirement violation. This lack coverage has been neglacted because of its few probability of occurrence. It may be handle by the redesigned of the safety mechanism.

Focus on one SW Safety Mechanisms: Watchdog Manager

Among the proposed safety mechanisms, it has been arbitrary decided to focus on the Watchdog Manager (WdgM) [14], which is the mechanism identified in Table 5.

The WdgM module is a key SW Module in the AUTOSAR-based architecture to ensure the application works safely, detecting 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: aliveness supervision, deadline supervision, and control flow supervision; and four error reactions: signaling errors to other AUTOSAR modules, logging the errors into Diagnostic Event Manager or Development Error Tracer modules, partition reset: restarting a specific OS-application, and micro-controller reset. The WdgM supervises the execution of the software by monitoring so called *supervised entities – SE*. These SEs have no fixed relationship with architectural building blocks in AUTOSAR, i.e. SWCs, CDDs, RTE, BSW modules…

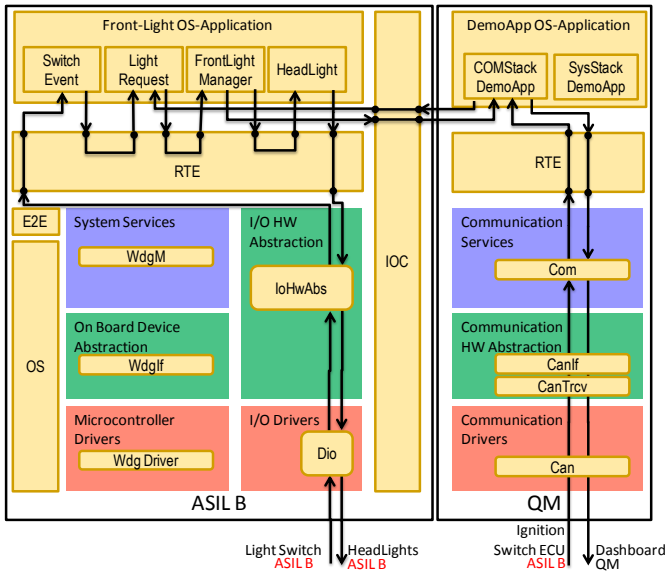

Figure 2. Software Architecture of the Front-Light Manager

In each SE, several supervision (alive, deadline, control flow) can be defined. Each SE is defined with checkpoints and transitions between checkpoints. It forms a graph of the SE, where checkpoints corresponds to the states of the graph. A graph may have one or more initial checkpoints and one or more final checkpoints. The graphs are defined statically during the configuration of the WdgM.

At runtime, the SEs call the WdgM API when they have reached a checkpoint. Then, the WdgM verify that the sequence of the received checkpoints, starting with any initial checkpoint and finishing with any final checkpoint, is correct.

Specifically, aliveness supervision is monitored by the call of one checkpoint by the supervised entity. The WdgM checks periodically that the checkpoint has been reached within given limits (not too frequent or not too rare). The deadline supervision checks the timing transition between two checkpoints of a SE. Finally, the Logical supervision monitors that the graph of a SE is executed correctly.

Table 5. Line Extracted from the Software FMECA Performed on the Software Architecture Described in Figure 2

item ID	Function Description	Failure Mode description	Local Effect	Product Effect without safety mechanism	Component #UE	ASIL	Safety mechanism	Product Effect with safety mechanism	Component #UE	ASIL
DemoApp_REQ_01	ComStackDemoApp must (periodically 10ms) read Ignition Position from COM through RTE (Non trusted)	Period too slow	Erroneous Ignition Position used	Erroneous inputs provide to Critical Application (Light Request SWC) (Service Calls Interferences)	SW-UE04	ASIL B	WdgM Aliveness supervision – supervised Entity 4 (10ms)	Reset + Safe Mode	SW-UE09	QM

Hence, it implies that every time the SE reports a new checkpoint, the WdgM must verify that a transition is configured between the previous checkpoint and the reported one.

A first version, implementing the AUTOSAR requirements, has been implemented (QM version). Then, safety analyses have been performed on this module, considering it as a SEooC.
The failure modes of this module are twofold: i) False alarm: the WdgM unexpectedly triggers a reconfiguration (signaling an error, logging an error, resetting the target…), ii) False negative: it is a bad coverage in the detection or the management of the error, i.e., the WdgM does not detect an erroneous behavior.

The safety analysis has highlighted a gap in the implementation of the WdgM as some errors could remain undetected.

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

Moreover, the safety analyses have revealed several potential causes of the WdgM failure modes, and safety mechanisms have been defined to mitigate them. E.g., a global variable was used to store the state of the WdgM, the corruption of this data was critical as it could lead to the failure modes. To prevent it, it was decided to triplicate its value in the memory, in order to avoid single point error in the design of the WdgM. This safety mechanism masks the error in one replication using a majority vote.
In order to handle these cases a second implementation of the WdgM has been realized. This "safety version" implements the requirements identified by the safety analyses.

4. Proof of Concept on the Case Study

In this section, fault injection tests' plan is illustrated and the results of these tests are described and analyzed. The tests focus on the SW FMECA line described in Table 5. The line enables to define two targets for the injection. First, the propagation of the considered failure mode may violate the FFI and may also lead to the violation of the SG1 or SG2. Secondly, it also identifies the WdgM as a mean to mitigate the critical path, and therefore to ensure the FFI and the safety requirements.

Even though it is not shown and stated explicitly all over the case study, all FI experiments have been defined following the FARM model.

Fault Injection Targets of Fault Injection

Verification of the Critical Path

The target of the experiment is the following item: "*ComStackDemoApp must periodically (10ms) read the Ignition Position from COM module through the Untrusted RTE"*. The failure mode "Period too slow" may lead to create interferences on the ASIL B *Light Request* SWC by providing an erroneous Ignition Position. ComStackDemoApp may interfere with the critical application of the Front-Light Manager. At the end, it may violate the safety goal SG1 and/or SG2 at system level.
For the safety requirement verification, 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.

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. An easy example is to kill the task responsible of the execution 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 20ms to 100ms with a step of 10ms.
Looking at the **A**ctivation model, it could be easily find that two use cases may lead to the violation of a Safety Goal. Indeed, the failure mode is critical when the ComStackDemoApp cannot provide the new value of the Ignition Switch to the IOC.
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 threw the CAN. In this case, if the COMStack_DemoApp period is too long, the headlight may light with a delay. Figure 3 illustrates this second use case in the case where the period is temporary double.

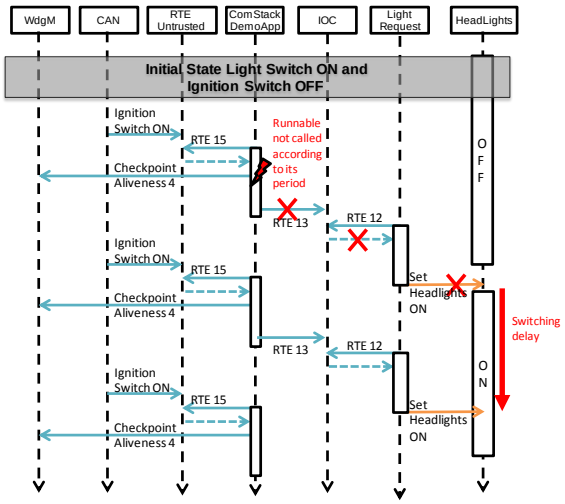

Figure 3. Simplified Sequence Diagram of the Activation of the Headlights.

The following **R**eadouts are needed to analyze the result of the experiment. The local effect of Table 5: "Erroneous Ignition Position used" should be monitored in the Light Request SWC but also the headlight state. Then, the WdgM should also be monitored to verify that it detects and handles the error. Finally, if the safety mechanism is efficient the headlights will be ON, otherwise they will remain OFF.

Safety Mechanisms

In Section 3, one safety mechanism has been proposed to handle this failure mode: using an aliveness supervision of the WdgM in order to detect it and then at first reset the application and if the error is not corrected put the system in a safe mode where the two headlights are always set ON. It has been shown, FI should also be used to assess the effectiveness of the WdgM coverage which has the same ASIL allocated than the Critical application: ASIL B.

The first **M**easures expected are the EDC and the ERC of the WdgM. The robustness of the WdgM implementation should also be assessed. In this case, the behavior of the WdgM, while its memories (RAM/ROM/stack) are corrupted, is assessed. This evaluates the quality of the code and could highlight several the weaknesses. On the WdgM case study, errors leading to false alarms and false negatives have both been found.
Finally, the same approach could be applied on each line of a FMECA, or by analyzing the critical paths identified.

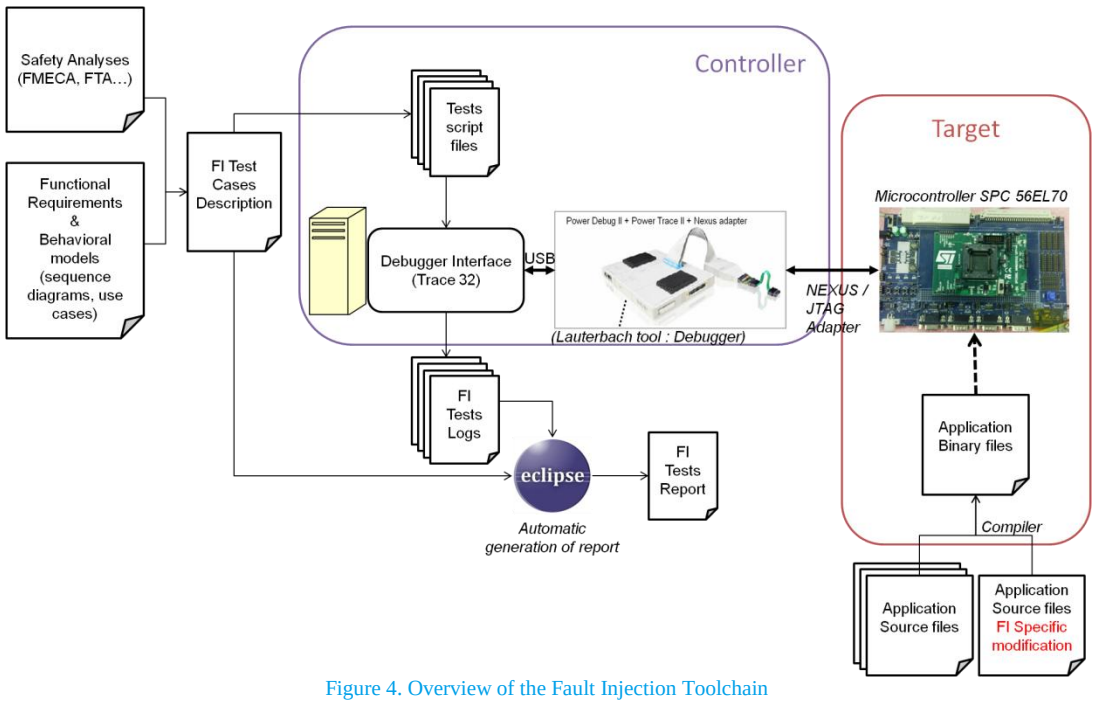

Figure 4. Overview of the Fault Injection Toolchain

Tests Platform Description

From a practical point of view, the FI tool developed at VALEO GEEDS provides all the functionalities required to implement fault injection. This platform uses two FI Techniques: Software-Implemented Fault Injection (SWIFI) and Nexus-Based Fault Injection. SWIFI is based on the modification of source code of the application. Nexus-based FI uses the on-chip debug capabilities of some micro-controllers in order to access the memories and corrupt these data.

Figure 4 shows an overview of the FI tool chain that encompasses: a target, a controller, the source code and an Eclipse-based application. The target is the Front-Light Manager application (binary files) running on the microcontroller SPC56EL70 describes in Section 3.

The Controller is composed of a Lauterbach debugger that is connected to the microcontroller throughout a Nexus Probe. The debugger enables to access all the memory sections of the microcontroller, particularly by monitoring the trace of execution of the application. It is also a fault injector, as it allows corrupting/modifying memory (this could be done on-the-fly during execution of the application). The debugger implements Nexus Class 3+ defined by the Nexus 5001 Forum™. This class of debugger enables two important features to perform fault injection without influencing the real-time execution of the application: "On-the-fly" runtime memory access and on-chip watch points. The watch point enables to trigger the fault injection or signal application events without stopping the application. The "On-the-fly" memory access capability solves the problem of writing or reading the memory without adding temporal overhead.
The Debugger is controlled by script files (PRACTICE Language) interpreted by Trace32 software that perform the interface with the probe.

The source code is compiled using Wind River to produce an .elf file (binary) that is then flashed onto the microcontroller. SWIFI can be implemented by modifying a part of the application source code that is then compiled and execute on the microcontroller. The Eclipse based application enables to automatically generate a report for all the tests executed on the platform. The inputs of the application are the FI tests cases description and the FI tests logs (readouts).

Tests Set-up

First, it is considered that the micro-controller and memories, are well protected against random faults by error detection and correction mechanisms implemented directly in the circuitery of the micro-controller.
Then, in order to prevent the other corruption, the first action before any fault injection experiment is to flash the application on the target. It loads the application and verifies that the loaded application is not corrupted.
Moreover, the fault injection tests cases mainly target global variables which are statically defined in the memory. Hence, the value of these variables can be efficiently controlled during the experiment.

Results Obtained on the Critical Path

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

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

- 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 = 20ms]
- Detected and tolerated errors: 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 = 30ms or 40ms]
- Detected and recovered errors: in these cases, the WdgM detects the aliveness error, and start a reset of the microcontroller. Then, in use case 1, there is a blinking where the headlight are OFF during 37ms (reset time) and then the system work properly

(headlights are ON). In use case 2, the system will not switch the system during a maximum of 220ms since the error is injected and the Ignition Status command received. It corresponds to the time of detection by the WdgM and reset of the application. [Periods between 50 and 100ms].

Test results qualification, i.e., the assessment of the readouts of the tests, could be challenging, see [15]. The behavior of an application may not be evaluated easily due to external environment complexity. However in the case of the Front-Light Manager, the complexity is low. The result qualification could be done observing identified output signals (e.g., headlights state) or selected parameters (detection flags) or exceptions.

All the tests could be put in cases α and δ of the Table 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 faulty behavior within the response time. Here, a single failure is not sufficient to lonely violate a safety goal, in the worst case, the failure mode occurs but reset is triggered fast enough to remain unnoticed.

The implemented safety mechanism is sufficient to handle the considered failure mode of Table 5.

Then, the same method should be performed for each failure mode identified in the FMECA.

Results Obtained on the Watchdog Manager

Evaluation of the Error Detection Coverage of the Two WdgM Implementations

The WdgM implementations, described in section 3, have been tested in order to evaluate their efficiency to detect and recover from Aliveness / Deadline / Control Flow errors. In other words, these tests aim at verifying WdgM implementations fulfill their functional specification when integrated. It also measures the improvements between the two versions

Here, the considered fault model is i) wrong checkpoints sent by a SWC to the WdgM and ii) checkpoints sent with wrong timing behavior.

To perform the tests, two instances of each supervision have been configured: one instance is enabled in the mode used for the tests and the second one is disabled in the mode used by the application. This enables to verify non-configured checkpoints, configured but not activated ones and activated ones. There are 256^2 possible checkpoints: there are 256 supervised with 256 checkpoints each.

The implemented configuration contains two checkpoints for the aliveness supervisions, 2 control flow supervisions with 6 checkpoints and 6 transitions each, and 2 deadline supervisions with 2 checkpoints each.

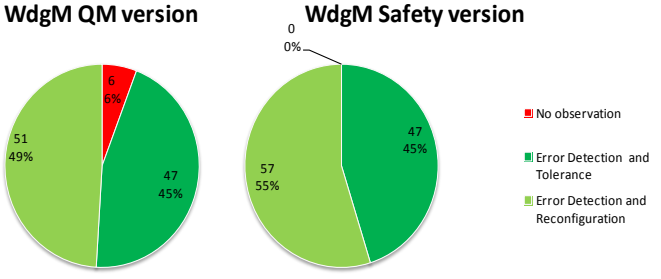

Figure 5. Effectiveness of the two WdgM implementations against the same fault model (104 tests cases)

In order to inject the fault, a SWC that mimics all the possible behaviors for checkpoints has been designed. Then, the internal variables of the WdgM and the resets of the microcontroller have been monitored. These variables help to verify if the intended supervision error has been detected, and the microcontroller reset enables to verify that the error has been recovered.

The results illustrated in Figure 5 have been obtained.

It can be observed that the QM version detects and recovers 94% of the tests cases (case α and β of the Table 1). This domain is constituted of two categories:

- A detected and tolerated error, i.e., the error is detected but do not necessitate resetting the microcontroller. It corresponds to tests where non-configured checkpoints are called. In this case, the WdgM returns that an error occurs, but performs no recovery.
- A detected error that reset the target. In this case the error is detected and the WdgM restart the microcontroller.

The 6% of "No Observation", i.e., the WdgM does not detect any errors and corresponds exactly to the cases identified in the safety analysis, (case γ and δ in Table 1). The results show these cases are handled by the safety implementation. Fault injection tests, here, improve the confidence in the Error Detection Coverage handled by the new implementation of the WdgM.

Considering the verification of the critical path of Table 5, these results highlight that the two implementations have no difference concerning the aliveness supervision. Hence, both could be used for to handle this failure mode.

Robustness of the Implementation of the WdgM

However, it is also important to verify that the implementation of the WdgM is robust. Indeed, the robustness of the WdgM against interferences from HW faults (bit-flip in the memory/registers) / Real-Time faults has to be assessed.

This assessment is important because the WdgM is a Component Off-The-Shelf (COTS); a robust implementation is needed in order to reuse it.

It is interesting to note the interferences on the WdgM may not directly impact the safety of the System, as its dysfunction are false alarms leading to a reset, however, interferences may be safety relevant with second order cut sets. In this case, an application error could be masked (no detections of deadline/aliveness/control flow errors), and a safety goal may be violated.

The fault model used for these tests is the corruption of variables of the WdgM. The variables have been identified based on the safety analyses performed on the WdgM. The chosen variables may, if they are corrupted, lead to a WdgM failure mode.

Two activation use cases have been identified. Firstly, the injection is intended to create a false alarm, therefore the application is in normal mode when the fault is injected. The second use case aims at showing a WdgM corruption may mask errors. Here, both alive, deadline and control flow errors have to be emulated. The activation of the WdgM error (variable corruption) is done before the alive/deadline/control flow error.

For these tests, the identified critical path and all the requirements could be used as oracle for the test cases. Then several variables and function calls along the critical path have been monitored using on-chip watch points of the debugger. Typically, these are the flags/variables that store the WdgM's detection of errors, the calls of recovery actions and the occurrences of resets. The objective is to verify that the reconfiguration is caused by the WdgM and that the right error has been detected.

Figure 6 shows the pie charts obtained with these tests. The results are categorized as follow:

- "No deviation Observed" corresponds to tests where the behavior of the WdgM is identical with or without the injected fault. E.g., the corrupted variable is refreshed with the correct value. (case α and δ of the Table 1).
- "Transient Internal Deviation Observed" corresponds to tests where the error propagates but is tolerated before reaching a failure mode of the WdgM. (case α and δ of the Table 1).
- "Internal latent error" corresponds to tests where no WdgM failure mode is reached, however, the corruption done or the propagation of this error remains latent and is not activated by the activation of the target. (Cases α and δ of the Table 1). These experiments correspond to the corruption of unused variables. In this category, the corruption done by fault injection experiments in WdgM memory is not activated and still remain at the end of the test (timeout expired). The memory remains corrupted but could be activated by other activation.
- "Failure mode reached". Here, a false alarm or a false negative is observed (case β and γ of the Table 1). Particularly, among the 76 tests that lead to reach a failure mode, alive/deadline/Control flow error (False negative) have not been detected in 15 tests.

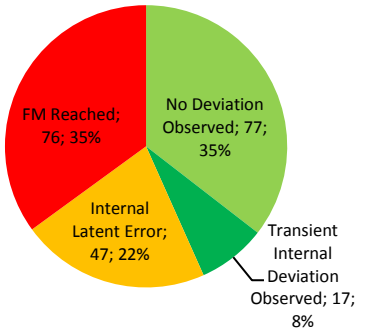

Figure 6. Robustness of the WdgM implementation (217 tests cases)

Results show that the test cases based on the safety analyses are efficient as there are only 35% of "No deviation Observed" tests.
They also highlight that some efforts should be done on the implementation focusing on basic events that lead to failure modes (red) and errors that remain latent (orange). For example, it could be proposed to triplicate some variables in order to protect it with majority vote. However, a tradeoff has to be made between memory consumption, performances and safety.

5. Summary/Conclusions

First, the investigation of the scope of FI has shown ISO 26262 requirements may impact significantly the development process of automotive critical system. Indeed, ISO 26262 requires fault injection for all ASIL levels for the verification and validation of failure propagation and mitigation by safety mechanisms.
Then, the objectives of FI have been discussed, and a method based on safety analyses has been proposed to define fault injection attributes of the FARM model: targets, measure, fault model, activation model and readouts.
The major advantage of this method is to rely on the previous activities of the development process. Hence, it helps to enhance the reliance on the safety analyses. Moreover, the method enables to keep the traceability of the safety requirements. This could be used to reduce the number of experiments.
However a main drawback is that the method required a high level of maturity of the safety analyses. Particularly, FMECA should be preferred to obtain better results, as FMECA are more complete analyses.
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.
Then FI tests have been planned and performed on a FI test platform.
A first interesting results obtained with the experiments' measure is the efficiency of the injected faults. Most of the injected faults have an impact on the target (they lead to a failure mode or they triggered a safety mechanism).
The obtained measures allow having reasonable proofs to demonstrate the effectiveness of the safety mechanisms (the WdgM) and the correct implementation of the safety requirements, and of 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.
Then, targeting standards safety mechanisms modules (particularly AUTOSAR ones) should be a priority, as they may be reused on multiple project. This also enables to define the fault injection tests, which should be done to benchmark different implementations of these modules.

6. References

1. ISO 26262 – Road Vehicles – Functional Safety, 10 November, 2011. http://www.iso.org/iso/home/news index/news archive/news.htm ?refid=Ref1499 [Online; accessed 30-Jul-2014].
2. L. Pintard, J.C. Fabre, M. Leeman, K. Kanoun, M. Roy, "From Safety Analyses to Experimental Validation of Automotive Embedded Systems," Dependable Computing (PRDC), 2014 IEEE 20th Pacific Rim International Symposium on , vol., no., pp.125,134, 18-21 Nov. 2014 doi: 10.1109/PRDC.2014.23
3. M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," Computer, vol. 30, no. 4, pp. 75–82, 1997.
4. P. Yuste, D. de Andres, L. Lemus, J. Serrano, and P. Gil, "Inerte: integrated nexus-based real-time fault injection tool for embedded systems," in Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on, pp. 669–669, 2003
5. D. Skarin, R. Barbosa, J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," Dependable Systems and Networks (DSN), 2010 pp. 557–562.
6. Giuffrida, C., Kuijsten, A., Tanenbaum, A.S., 2013. EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments, Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim Int. Symp. on, pp. 31-40.
7. N. Silva, R. Barbosa, J.C. Cunha, M. Vieira, "A view on the past and future of fault injection," Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on , vol., no., pp.1,2, 24-27 June 2013 doi: 10.1109/DSN.2013.6575332
8. F. Ayatolahi, B. Sangchoolie, R. Johansson, and J. Karlsson, "A study of the impact of single bit-flip and double bit-flip errors on program execution," in Computer Safety, Reliability, and Security (F. Bitsch, J. Guiochet, and M. Kaaniche, eds.), vol. 8153 of Lecture Notes in Computer Science, pp. 265–276, Springer Berlin Heidelberg, 2013.

9. Islam, M. M., Karunakaran, N. M., Haraldsson, J., Bernin, F., & Karlsson, J. (2014, May). Binary-Level Fault Injection for AUTOSAR Systems (Short Paper). In Dependable Computing Conference (EDCC), 2014 Tenth European (pp. 138-141). IEEE.
10. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," IEEE Trans. on Software Engineering, vol. 16, no. 2, pp. 166–182, 1990.
11. Datasheet SPC56EL70 32-bit Power Architecture® microcontroller for automotive SIL3/ASILD chassis and safety applications, DocID023953 Rev 4, October 2013
12. AUTOSAR Development Cooperation, http://www.autosar.org [Online; accessed 30-Jul-2014].
13. AUTOSAR Specification of Operating System V5.3.0 R4.1 Rev 3, https://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf [Online; accessed 30-Jul-2014]
14. AUTOSAR Specification of Watchdog Manager V2.2.0 R4.0 Rev 3, https://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/system-services/standard/AUTOSAR_SWS_WatchdogManager.pdf [Online; accessed 30-Jul-2014].
15. D. Trawczynski, J. Sosnowski, P. Gawkowski , Analyzing fault susceptibility of ABS microcontroller, *Computer Safety, Reliability, and Security, Springer*, 2008, 360-372

7. Acknowledgments

The authors thank Youness Kamel for his contribution on the WdgM experiments, and also SAE reviewers for their insightful comments.

8. Definitions/Abbreviations

API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
BE	Basic Event
BSW	Basic Software
CAN	Controller Area Network
CPA	Critical Path Analysis
COTS	Component Off-The-Shelf
EC	Equivalence Class
ECU	Electronic Control Unit
EDC	Error Detection Coverage
ERC	Error Recovery Coverage
FFI	Freedom From Interferences
FI	Fault Injection
FM	Failure Mode
FME(C)A	Failure Mode Effects (and Criticality) Analysis
FTA	Fault Tree Analysis
IO	input or output
IOC	Inter OS-Application Communicator
ISO	International Standard Organization
MMU	Memory Management Unit
MPU	Memory Protection Unit
OS	operating system
PHA	Preliminary Hazard Analysis
QM	Quality Management
RTE	Run-Time Environment
SE	supervised entity
SEooC	Safety Element out of Context
SG	safety goal
SWC	Software Component
SWIFI	Software Implemented Fault Injection
UE	undesired event
WdgM	Watchdog Manager