POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Development of a fault injection methodology and fault coverage analysis for a safety-relevant block

Supervisors
Matteo SONZA REORDA
Mariagrazia GRAZIANO
Carlo RICCIARDI

Candidate
Alfredo PAOLINO

Internship supervisors
Maria Silvia RATTO
Alberto RANERI

Academic Year 2022-2023

♪ *Ora posso correre*
e giocare ♪

Abstract

As electronic products become integral to daily life, their presence in safety-critical automotive systems becomes pervasive. The International Standard Organization (ISO) 26262 standard outlines guidelines for establishing a good level of safety in automotive System-on-Chips (SoCs) but, despite the adherence to this standard, operational failures still remain possible, prompting the need for Safety Mechanisms (SMs) to reduce random hardware faults.

Nevertheless, SMs must go through a fine verification and validation phase, often involving fault injection, to ensure that they are compliant with the standard and they work as intended.

This thesis examines the unique challenges posed by safety-critical SoCs and their SMs within the automotive sector. In particular, it proposes a comprehensive in-house fault injection and analysis tool named *Fault Injection e Verification Component* (*FIeVC*) to ease the development of safety-critical components in Bosch SensorTec (BST), highlighting its benefits and potential integration challenges. The *FIeVC* tool is developed in Specman *e* and leverages parallel fault simulation to compare a fault-free system and a faulty counterpart for mismatch detection, using four main components: the Monitor, the Sequencer, the Bus Functional Model (BFM) and the Analyzer.

The thesis also details the development of preliminary scripts needed for generating fault lists and modifying testbenches, both crucial aspects of a successful fault simulation. The tool is evaluated using an accelerometer datapath from a Micro Electro-Mechanical Systems (MEMS) Inertial Measurement Unit (IMU) platform developed by BST.

The tests conducted to assess the *FIeVC* performance, show that the tool takes 68% less time to perform a full fault simulation compared to traditional tools while keeping the Test Coverage (TC) level basically unchanged. These substantial simulation time improvements are mainly obtained by making use of the parallel algorithm and the advanced Specman *e* Testflow feature to implement the most critical and time-consuming task, the Design Under Test (DUT) reset procedure.

Contents

List of Figures IV

Acronyms V

Introduction 1

1 Background 7
1.1 Fault life cycle . . . 7
1.2 Fault Models . . . 8
1.2.1 Stuck-At Faults . . . 8
1.3 Functional safety in ISO 26262 . . . 10
1.3.1 Failure modes classification . . . 10
1.3.2 Safety metrics . . . 11
1.3.3 Automotive Safety Integrity Level (ASIL) . . . 12
1.4 Safety Mechanisms (SMs) overview . . . 13
1.5 Micro Electro-Mechanical Systems (MEMS) . . . 15
1.5.1 Accelaration sensor . . . 16
1.5.2 Gyroscope sensor . . . 16
1.5.3 Bosch SMI240 Inertial Measurement Unit (IMU) . . . 18
1.6 Related works . . . 19

2 *Fault Injection e Verification Component* (*FIeVC*) 21
2.1 Generic parallel fault simulator structure . . . 22
2.2 Preparatory scripts . . . 23
2.2.1 Fault list generation script . . . 23
2.2.2 TestBench (TB) readaptation script . . . 24
2.3 *FIeVC* structure . . . 26
2.3.1 Fault item . . . 27
2.3.2 Signal map . . . 27
2.3.3 Monitor . . . 27
2.3.4 Analyzer . . . 28
2.3.5 Sequencer . . . 30
2.3.6 Bus Functional Model (BFM) . . . 30
2.4 Simulation flow . . . 30

2.5 Resetting the DUT using the Testflow feature . . . 32
2.5.1 Simulation flow using the Testflow feature . . . 33
2.6 Backpropagation issue . . . 34
2.6.1 External backpropagation . . . 34
2.6.2 Internal backpropagation . . . 35

3 Xcelium Fault Simulator (XFS) 37
3.1 Main features . . . 37
3.2 Simulation flow . . . 38

4 Results 41

5 Conclusions 43

List of Figures

1 Typical bathtub curve representing the Failure Rate (FR) vs. Time graph . 2
2 Dependability and Security tree 2
1.1 The fault evolution inside a system 7
1.2 Stuck-At 0 (SA0) example 9
1.3 Stuck-At 1 (SA1) example 9
1.4 Flow diagram for failure mode classification [13] 11
1.5 ASIL classification table [32] 12
1.6 Target values to obtain a certain ASIL classification [19] 13
1.7 SMs timing quantites 14
1.8 Size comparison between a MEMS sensor and 1 euro cent coin 15
1.9 MEMS sensor module internal structure 15
1.10 MEMS accelerometer principle for x-axis and y-axis 16
1.11 Effect of Coriolis force on a tuning fork gyroscope 17
1.12 MEMS gyroscope mechanical structure 17
1.13 Cascaded Integrator Comb $(CIC)^2$ filter structure for consumer electronics applications 18
1.14 CIC^2 filter structure for automotive applications 19
2.1 Parallel fault simulator diagram 22
2.2 Functional unit representation (credits to Cadence®) 25
2.3 Simplified representation of the "mismatch_detected" logic 25
2.4 Simplified representation of the "fail_detected" logic 25
2.5 *FIeVC* structure with simplified connections 26
2.6 *FIeVC* fault classification (credits to Cadence®) 28
2.7 *FIeVC* fault classification flow 29
2.8 *FIeVC* fault simulation flow 31
2.9 *FIeVC* fault simulation flow with active Testflow feature 33
2.10 *force* command propagation error 34
2.11 Extended TB structure to solve external backpropagation issue 35
3.1 XFS fault simulation flow (credits to Cadence®) 38
4.1 Comparison between XFS and *FIeVC* fault classifications 41
4.2 Comparison between XFS and *FIeVC* TC and total simulation time 42

Acronyms

SoC System-on-Chip

ABS Anti-lock Braking System

ESC Electronic Stability Control

ACU Airbag Control Unit

ISO International Standard Organization

E/E Electrical/Electronic

FR Failure Rate

SM Safety Mechanism

DC Diagnostic Coverage

SPFM Single Point Fault Metric

LFM Latent Fault Metric

PMHF Probabilistic Metric for Hardware Failure

BST Bosch SensorTec

DUT Design Under Test

EDA Electronic Design Automation

KPI Key Performance Indicator

FIeVC *Fault Injection e Verification Component*

XFS Xcelium Fault Simulator

RE Regular Expression

TLM Transaction Level Modeling

SG Safety Goal

ASIL Automotive Safety Integrity Level

MEMS Micro Electro-Mechanical Systems

IMU Inertial Measurment Unit

ASIC Application Specific Integrated Circuit

BFM Bus Functional Model

RTL Register-Transfer Level

SAF Stuck-At Fault

SA0 Stuck-At 0

SA1 Stuck-At 1

CAT Cell-Aware Test

SEU Single Event Upset

SET Single Event Transient

IC Integrated Circuit

FS Functional Safety

FMEDA Failure Modes Effects and Diagnostic Analysis

ECC Error Correction Code

DCLS Dual Core Lock Step

TMR Triple Modular Redundancy

LBIST Logic Built-In Self-Test

FDTI Fault Detection Time Interval

FRTI Fault Reaction Time Interval

FHTI Fault Handling Time Interval

FTTI Fault Tolerant Time Interval

IMU Inertial Measurement Unit

LGA Line Grid Array

CRC Cyclic Redundancy Check

FIT Failure In Time

BFM Bus Functional Model

CIC Cascaded Integrator Comb

ADC Analog-to-Digital Converter

DMR Double Modular Redundancy

FPGA Field Programmable Gate Array

UVM Universal Verification Methodology

TB TestBench

TCL Tool Command Language

CLI Command-Line Interface

UU Unpropagated Undetected

UD Unpropagated Detected

DU Dangerous Undetected

DD Dangerous Detected

TC Test Coverage

Introduction

Today, electronics have become part of our everyday lives. Consumer electronics, telecommunications, and avionics are just a few of the numerous fields in which electronics is a leading portion of the final product.
The automotive sector is for sure no less, with an estimation of around 1,000 SoCs mounted on a traditional vehicle and at least twice this number on an electric vehicle today. Nevertheless, there is a significant distinction between an automotive SoC and a smartphone SoC, which has to do with the potential implications of a failure in any of these devices. A large majority of automotive SoCs are in fact defined as "safety-critical".
Safety-critical chips must be designed to operate in systems where the chip's failure can seriously harm human life, property, or the environment. Therefore, a failure in a safety-critical chip can have catastrophic consequences, such as accidents, injuries, or even fatalities. On the other hand, failures in normal application chips typically have less severe consequences, often limited to operational disruptions or inconvenience.
Anti-lock Braking System (ABS) controllers, Electronic Stability Control (ESC) systems, or Airbag Control Units (ACUs) are only some examples of safety-critical systems mounted on a car, and it is straightforward to imagine what consequences a single failure in one of these systems could bring.
In an effort to limit chip failures and any regrettable incidents associated with them, the automotive industry adopted in 2011 the so-called ISO 26262 safety standard, which focuses on managing and minimizing risks during the entire lifecycle of a safety-critical SoC for road vehicles. The standard was updated in 2018.
ISO 26262 specifies a set of guidelines to follow and activities to perform during the development process to ensure the achievement of Functional Safety (FS), defined as the "absence of unreasonable risk due to hazards caused by malfunctioning behavior of Electrical/Electronic (E/E) systems [17]".
Yet, even if both the specification and design phases are carried out in compliance with the ISO 26262 standard, an SoC is never 100% safe due to some failures that could arise during the operational phase. Thus, the operational phase of an electronic device is typically split into three distinct periods, strictly related to the Failure Rate (FR) of the component itself [31]:

- The "early failure" period, in which possible manufacturing defects present in the component rise and cause malfunctioning. These kinds of failures are labeled as *systematic* and this phenomenon is known as infant mortality. During this period the FR decreases over time.

- The "intrinsic failure" period, in which systematic defects are less of a threat, but random hardware faults could still arise due to different reasons, such as extreme environmental conditions of temperature, humidity or dust, exposure to liquids, Electro-Static-Discharge, Electromigration effect and so on. During this period the FR remains basically constant but higher than 0.
- The "wear-out" period, in which the component starts showing signs of failure due to aging and wear-out. During this period the FR starts to rise up to a point where it is no longer acceptable and the component must be replaced.

The FR can be plotted on an FR vs. Time graph, obtaining the so-called *bathtub* curve (Figure 1).

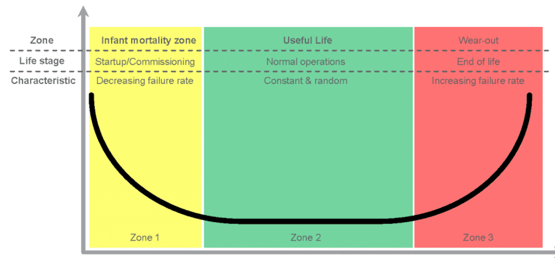

Figure 1. Typical bathtub curve representing the FR vs. Time graph

The target of any semiconductor company is then to reduce the frequency and severity of random hardware faults, extending the useful life of the chip as a result.
To understand what means a company has to pursue this goal, the concept of safety must be extended to the broader concept of dependability, defined as "the ability to deliver a service that can be justifiably trusted" and exhaustively explained and discussed in [9]. In this paper, the taxonomy of a dependable system is described as a tree (figure 2).

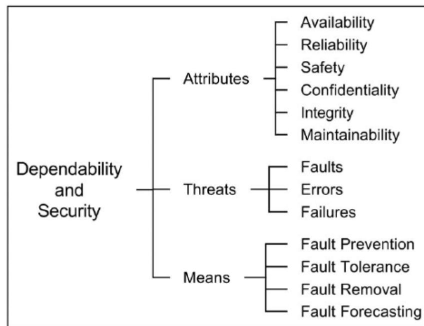

Figure 2. Dependability and Security tree

As shown in figure 2, safety is one of the attributes of dependability, and the means to ensure dependability are four, each of them defining different techniques complementary

to each other.
Among them, the focus of this thesis targets fault tolerance, which establishes procedures enabling a system to continue providing the required service in the presence of faults, at the cost of possible degradation of the service itself. Fault-tolerant systems are then equipped with Safety Mechanisms (SMs), needed to achieve the optimal robustness to failures. SMs are able to mitigate the rate of random faults, contain their consequences and reach the established safety goals.
A careful analysis should be done to understand which SMs best fit the design, keeping in mind their pros and cons in terms of effectiveness and cost. The best SMs setup is the one that optimizes the trade-off between these two elements while being compliant with safety requirements and standards.
Regardless of the SMs inserted, ISO 26262 states that a validation phase must be done before going into production. Validating these mechanisms involves verifying their effectiveness, reliability, and adherence to safety standards and requirements.
Most common validation approaches rely upon analytical techniques, prototyping, compliance testing and fault injection. And precisely the latter is the topic on which this thesis revolves.
The basic concept behind fault injection is self-explanatory; it consists of intentionally introducing faults in the chip or its model to simulate failure scenarios and evaluate the response of SMs. Whatever the metric, the fault model or the behavioral domain of the component taken into account [20], a fault simulator is essential to perform the evaluation. Today, many different commercial fault simulators are already present on the market. Among them, the Xcelium Fault Simulator (XFS) from Cadence® operates within the Xcelium™ Simulator compiled engine, which is the standard simulator used in BST for simulation and verification processes. This means that with very little effort a fault simulation campaign could be performed on the Design Under Test (DUT), making it a very attractive solution for BST.
Nevertheless, these Electronic Design Automation (EDA) tools quite often require expensive licenses and, for companies with many employees, many licenses would be required. Thus, the high cost could cause a company to give up using them and look for different solutions.
The ultimate goal of this thesis work is to study all the fundamental concepts of fault injection and explore the benefits that an in-house fault injection and analysis tool could bring in BST during the development of safety-critical components. Pros and cons of this solution are analyzed from many points of view, taking into account parameters like performance, flexibility, accuracy of results and integration with the underlying verification environment.
BST makes large use of the Specman *e* Hardware Verification Language for their verification environments, thus also the tool is developed in *e* to guarantee full compatibility, and named "*Fault Injection e Verification Component* (*FIeVC*)".
Several algorithms exist to perform fault simulation, but all of them require a fault list in addition to the description of the design and the input stimuli. On top of that, the *FIeVC* uses the *parallel* algorithm, which also needs a modified testbench, with two DUTs instead of one. This algorithm consists, in fact, of simulating at the same time both a fault-free machine and a faulty machine, checking for possible mismatches in their outputs. To

ease and automate the process of generating the fault list and duplicating the DUT, the preliminary part of this thesis work mainly aims at developing two scripts:

- A TCL script is used to generate the circuit fault list. This script allows the user to customize the fault list and perform injection recursively on the entire DUT or only some specific sub-components.
- A Python script is used to make the testbench suitable for fault injection. Through a precise parsing of both the testbench file and the DUT file, this script duplicates the DUT for the parallel fault simulation, generates new testbench signals for mismatch detection and connects them to the *FIeVC* via a Transaction Level Modeling (TLM) interface.

These two scripts make extensive use of Regular Expressions (REs) and leverage on BST naming conventions to increase their flexibility and adaptability to different components at different hierarchy levels.
For what concerns the core of the thesis, its focus is of course on the *FIeVC* development. The *FIeVC* is composed of 4 main components:

- *Monitor*, which constantly monitors the mismatch signals to detect possible discrepancies between the golden and the faulty outputs.
- *Sequencer*, which generates the fault that is going to be injected next.
- *BFM*, which acts as a driver and forces the fault inside the DUT.
- *Analyzer*, which receives the results obtained from the monitor to categorize the fault and extract a report file.

One of the greatest features of the *FIeVC* is that it works in a completely transparent manner w.r.t the pre-existing verification environment and it can be easily connected or disconnected by defining or not an environment variable.
Finally, the last part of the thesis aims at performing a fault simulation campaign in XFS, to compare and validate the results obtained from the *FIeVC*. This simulation is carried out by injecting the same faults into the design to have a 1-to-1 comparison and check for possible discrepancies.
Benchmark tests to assess the *FIeVC* performance have been performed using as DUT the accelerometer datapath of the *SMI240*, a MEMS IMU platform developed for the automotive market.
A testing fault simulation, performed with a 1,000-fault sample, shows that the *FIeVC* is able to obtain results very similar to XFS while speeding up the simulation process by three times.
Lastly, the user experience has been extremely simplified by using a Makefile which hides the complexity of the flow. In detail, an already existing Makefile has been expanded with three new make targets through which the setup phase, the simulation phase (with the *FIeVC* or XFS) and the comparison phase can be launched with a simple command.
The thesis is organized as follows: Chapter 1 provides some technical background to better understand the proposed work and describes the state-of-art for what concerns

FS and fault simulation tools. Chapter 2 describes in detail the *FIeVC* itself and the preliminary phase scripts. Chapter 3 explains the steps done to perform a complete fault simulation campaign through XFS, in order to have a basis for comparison with a commercial fault simulation tool. Chapter 4 shows the results obtained with the *FIeVC*, and compares them with those obtained through XFS. Finally, the thesis ends with Chapter 5, which illustrates further implementations and ideas for the next future and proposes the concluding remarks.

Chapter 1

Background

1.1 Fault life cycle

Alongside dependability means, the dependability tree also defines dependability threats: faults, errors and failures. These threats are closely related to each other.
A *fault* is defined as the presence of a defect inside the system. If the fault is excited by the correct stimuli, it starts spreading its malicious effect all around the circuit, becoming an *error*. If the error propagates deeply into the circuit, making its way up to the functional outputs, it becomes a *failure*. A failure (also called misbehavior) is therefore an instance in time when an output behavior not compliant with the circuit specifications is observed. The failure is then a manifestation of the fault to the user.
The life cycle of a fault is thus divided into three phases:

- *Quiescent* phase. In this phase, the fault is present in the system but, due to the lack of stimuli, it has not been propagated yet. The fault is said to be *dormant* or *passive*.
- *Activation* phase. In this phase, the fault is activated evolving into an error.
- *Propagation* phase. In this phase, the fault propagated to the circuit outputs, evolving into a failure and causing misbehaviors.

At the system level, misbehaviors are the cause of hazards, generating in turn harmful situations (figure 1.1).
When performing fault simulation, the goal is then to excite (fault→error) and observe (error→failure) the highest amount of faults, to check which of them could be dangerous.

Figure 1.1. The fault evolution inside a system

1.2 Fault Models

Several physical sources can be the cause of hardware faults. Some examples of real defects in chips are:

- Manufacturing defects, like missing contacts, misaligned masks, presence of dust or impurities during fabrication.
- Material defects, like imperfections in the crystal structure of silicon, wrong doping and cracks.
- Time-dependent defects caused by aging, dielectric breakdown, electromigration.

Nevertheless, the simulation of these kinds of faults would be a quite challenging task in the early phases of the design process. In fact, it would be practically impossible to inject a physical error in a purely abstracted description of the circuit like the Register-Transfer Level (RTL) or the gate-level one.
Yet, it is possible to cluster the physical faults in fault models, to enable the possibility of performing fault simulation also on these abstracted descriptions.
A fault model is a logical representation of the physical fault, based on the effect that the fault would have on the circuit.
These kinds of faults become then injectable because there is a precise correspondence between the effect caused by them on the physical chip and on its logical descriptions.
If two different physical faults have common properties, and a similar manifestation in terms of obtained results, they can be modeled by the same fault model, widely reducing the variety of faults and in turn the complexity of the fault simulation process.
Some of the most popular fault models are:

- Stuck-At Fault (SAF) model, more precisely SA0 and SA1
- Bridging fault model
- Transistor fault model, more precisely stuck-open and stuck-short
- Delay fault model
- Cell-Aware Test (CAT) fault model
- Transient fault model, more precisely Single Event Upset (SEU) and Single Event Transient (SET)

1.2.1 Stuck-At Faults

SAF model is one of the most common fault models because it covers a large majority of physical defects and manufacturing errors occurring in an Integrated Circuit (IC). These faults are characterized by a specific type of electrical behavior in which a signal line becomes stuck at a particular logic value (either high or low) regardless of the input conditions. When a SAF rises on a net, the data carried on it is forced to assume the logic-0 (SA0) value or the logic-1 (SA1) value.

In a circuit having n nodes, it is possible to model:

- $2n$ single SAFs.
- 3^n-1 multiple SAFs.

In complex designs, the amount of multiple SAFs would make the analysis extremely complex and time-consuming, but fortunately, it is possible to infer the behavior of multiple SAFs from single SAFs tests [8, 23]. This enables the possibility of restricting the analysis to single SAFs only.
The most common technique to test SAFs is called "path sensitization" [14, 37]. Without digging too much into the theory and the math behind this technique, it is worth knowing that it is based on two consequential concepts:

- Fault *excitation*, which requires creating a difference between the good and the faulty circuit in the fault site (i.e. drive a logic-1 on a SA0 net and vice-versa).
- Fault *observation*, which requires propagating this difference up to an output.

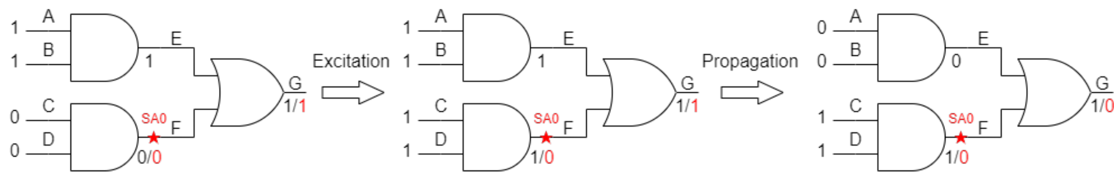

Figure 1.2. SA0 example

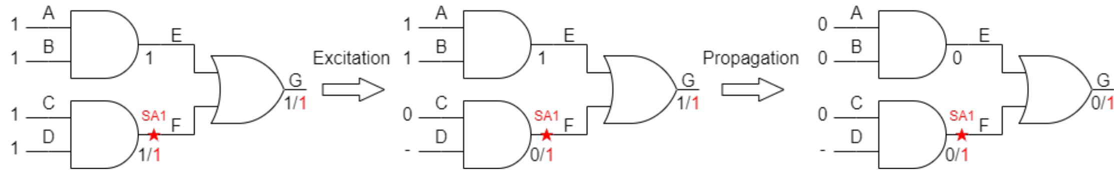

Figure 1.3. SA1 example

Figures 1.2 and 1.3 give a clear example of what excitation and propagation mean.
The upstream gates need to have inputs capable of creating a difference in value between the good circuit and the faulty circuit for excitation to occur. On the other hand, the downstream gates should not interfere with the faulty node's value during propagation, allowing the output value to reflect solely the value present on the faulty node.

1.3 Functional safety in ISO 26262

To have a measure of how much a hardware component is able to achieve its Safety Goals (SGs), ISO 26262 states that both qualitative and quantitative verification via various metrics must be done. From the scores obtained in each of these metrics, an ASIL is awarded to the hardware component.

1.3.1 Failure modes classification

Before diving into the safety metrics defined in ISO 26262, it is important to understand how failure modes, and in turn faults, can be classified. Failure modes are in fact "the manners in which an item fails to provide the intended behavior" [18] i.e. the ways in which a fault can manifest itself.
So, depending on the effects that faults produce on the circuit, they fall into one of these four different categories [13, 15] (figure 1.4):

- *Safe* faults. They do not increase the probability of violation of a SG because they are propagated neither to functional nor to safety outputs.
- *Single-point* faults. They are propagated to a functional output, but not covered by any SM, directly leading to the violation of a SG.
- *Residual* faults. They are propagated to a functional output and covered by a SM, but this is not able to prevent the violation of a SG.
- *Multi-point* faults. They are propagated to a functional output and covered by a SM, which is able to detect and correct them. The only moment in which they can cause the violation of a SG is when another fault, in the SM itself, is present. They are in turn subdivided in:
  - *Detected*. Even in the presence of multiple faults, the SM is able to detect and correct them.
  - *Perceived*. In the presence of multiple faults, they do not cause the violation of a SG but somehow impact the driving experience.
  - *Latent*. They are corrected by the SM without being detected.

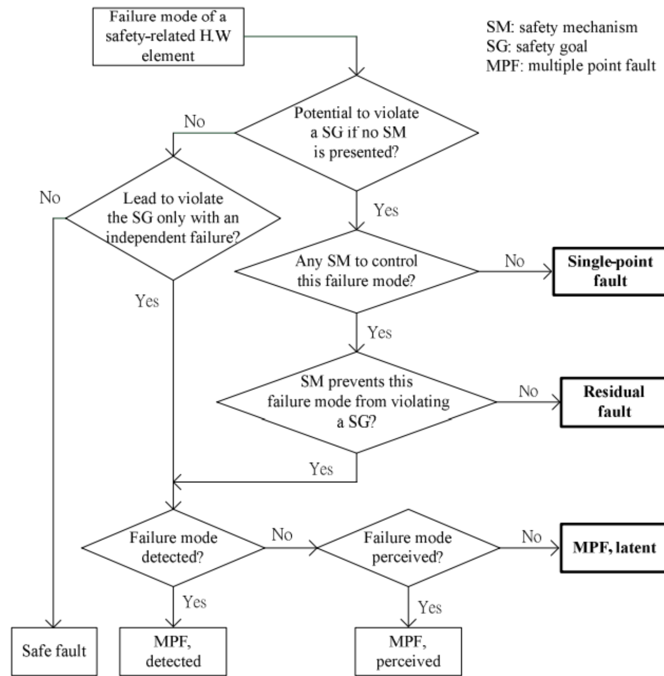

Figure 1.4. Flow diagram for failure mode classification [13]

1.3.2 Safety metrics

After the failure modes are classified three key metrics are used to evaluate the effectiveness of a safety architecture:

- Single Point Fault Metric (SPFM). It is a hardware architectural metric that reflects the robustness of a safety design to deal with single-point and residual faults. It reveals whether or not the SMs present in the component are enough to prevent the risk from single point and residual faults in the hardware architecture.

- Latent Fault Metric (LFM). It is a hardware architectural metric that reflects the robustness of a safety design to deal with latent faults. It reveals whether or not the SMs present in the component are enough to prevent the risk from latent faults in the hardware architecture.

- Probabilistic Metric for Hardware Failure (PMHF).Is the sum of the single-point, residual and multi-point fault metrics. Its unit of measure is the Failure In Time (FIT) (1 FIT = 1 failure ever 10^9 hours).

All these metrics strongly depend on the Diagnostic Coverage (DC) of each SM implemented in the system. The DC is in fact the ratio of detected/controlled failures over the total failures and it measures the effectiveness of the SM [3].

1.3.3 Automotive Safety Integrity Level (ASIL)

The meaning of ASIL is two-fold.
When the ASIL is related to operating scenarios, it represents a risk classification scheme, ranging from grade A (the least stringent) up to grade D (the most stringent); the higher the ASIL classification, the higher the injury risk. The ASIL classification of a scenario is performed through a risk assessment that takes into account three parameters: severity, exposure and controllability of hazardous events. These three parameters are classified in a range from 1 to 4, and the product between them determines the ASIL [5] (figure 1.5).

$$\text{ASIL} = Severity\ x\ (Exposure\ x\ Controllability)$$

Severity	Exposure	Controllability		
		C1 (Simple)	C2 (Normal)	C3 (Difficult, Uncontrollable)
S1 LIGHT AND MODERATE INJURIES	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	QM
	E3 (Medium)	QM	QM	A
	E4 (High)	QM	A	B
S2 SEVERE AND LIFE THREATENING INJURIES – SURVIVAL PROBABLE	E1 (Very low)	QM	QM	QM
	E2 (Low)	QM	QM	A
	E3 (Medium)	QM	A	B
	E4 (High)	A	B	C
S3 LIFE THREATENING INJURIES, FATAL INJURIES	E1 (Very low)	QM	QM	A
	E2 (Low)	QM	A	B
	E3 (Medium)	A	B	C
	E4 (High)	B	C	D

QM (Quality Management)
Development supported by established Quality Management is sufficient.

A **lowest ASIL**
Low risk reduction necessary
B
⋮
C
D **highest ASIL**
High risk reduction necessary

Figure 1.5. ASIL classification table [32]

On the other hand, when the ASIL is related to hardware components, it is an attestation that the component meets both qualitative and quantitative expectations in terms of SGs for that ASIL level [4]. This means that an ASIL C component is suitable for ASIL A, B and C (but not D) applications.
For a component to be attested with a certain ASIL classification, it must meet precise requirements for each of the above-mentioned metrics (figure 1.6).

	ASIL B	ASIL C	ASIL D
SPFM	> 90%	> 97%	> 99%
LFM	> 60%	> 80%	> 90%
PMHF	< 100 FITs	<100 FITs	< 10FITs

Figure 1.6. Target values to obtain a certain ASIL classification [19]

1.4 Safety Mechanisms (SMs) overview

For safety-critical systems, the role of a safety engineer is to devise a FS architecture that includes SMs to meet the required metrics (SPFM, LFM, PMHF). These SMs are additional hardware components, able to detect or correct errors and completely independent from the functions they protect to achieve optimal robustness to failure. Usually, a Failure Modes Effects and Diagnostic Analysis (FMEDA) drives the design exploration, identifying where to focus the design effort to meet the constraints.
The selection of the best SM for a specific building block or system needs a careful analysis of the trade-off between the effectiveness for safety metrics and the cost; the effectiveness is represented by the DC, while the cost is represented by power consumption, area overhead, performance impact, and even verification time and automation. Usually, high effectiveness increases the cost in direct proportion, but to optimize this trade-off, the benefits and costs of SMs should be analyzed in all aspects.
To this end, it is possible to classify SMs in four different categories, based on their working principles:

- *Information redundant.* In this case, additional data are added to protect the original information. Examples of this class are parity bits and Error Correction Codes (ECCs).

- *Time redundant.* In this case, the same operation is executed more than once on the same functional unit, but at different times. An example of this class is the Execute-Retry-Checkpointing-Recovery technique.

- *Space redundant.* In this case, the same operation is executed on multiple functional units simultaneously. Examples of this class are Dual Core Lock Step (DCLS) and Triple Modular Redundancy (TMR).

- *Diagnostic.* In this case, the SM directly verifies if the intended functionality is working correctly. An example of this class is the Logic Built-In Self-Test (LBIST) technique.

While the hardware architectural metrics discussed in 1.3.2 do not encompass timing constraints, it is evident that assessing the safety mechanisms comprehensively requires considering timing performance. Some temporal quantities are then specified for both the system and the SM in order to check if the latter behaves correctly [26]:

- Fault Detection Time Interval (FDTI). It is a property of the SM itself, defined as the time elapsed from the occurrence of a fault to its detection.
- Fault Reaction Time Interval (FRTI). Again, a property of the SM, defined as the time elapsed from detecting a fault to recovering and restoring a safe state.
- Fault Handling Time Interval (FHTI). The sum of FDTI and FRTI.
- Fault Tolerant Time Interval (FTTI). It is both a property and a requirement of the system, defined as the time within which it should be capable of detecting the presence of faults and transitioning to a safe state. Failure to meet this timing requirement could result in the fault escalating into a hazard and compromise the correct behavior of the system.

To be compliant with the specifications, the property FHTI $\leq$ FTTI must hold for every SM present in the system.

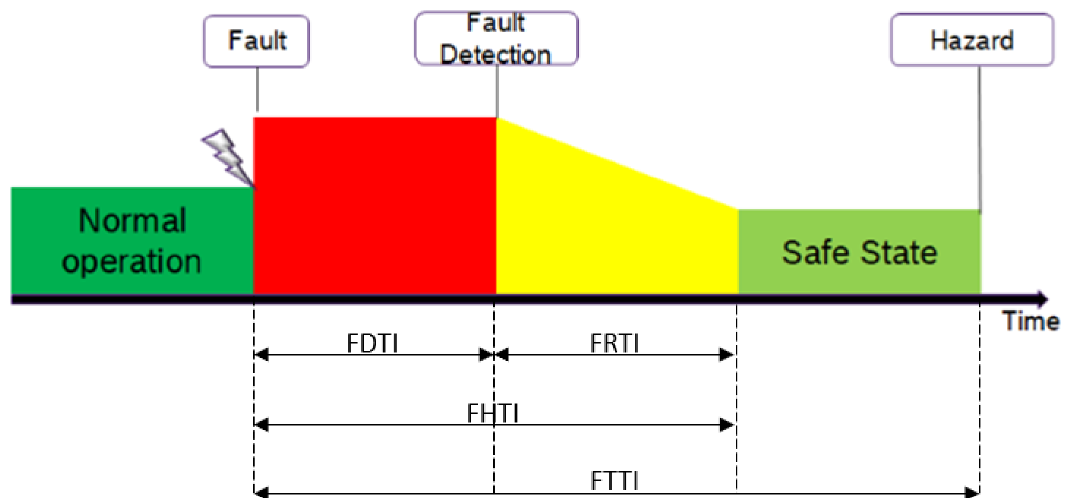

Figure 1.7. SMs timing quantites

1.5 Micro Electro-Mechanical Systems (MEMS)

MEMS is a process technology used to create tiny integrated devices or systems that combine mechanical and electrical components. MEMS fabrication uses semiconductor manufacturing processes like deposition, doping or photolithography, meaning that both the mechanical parts and the electronics that control them can be built on the same silicon chip. These devices can range in size from a few micrometers to millimeters (figure 1.8) and have the ability to sense, control and actuate on the micro scale while generating effects on the macro scale. The MEMS sensor element and the Application Specific Integrated Circuit (ASIC) evaluation circuit are both crammed inside a semiconductor housing (like a Line Grid Array (LGA)) to form a MEMS sensor module, isolated from the external environment through a mold compound (figure 1.9) [36].

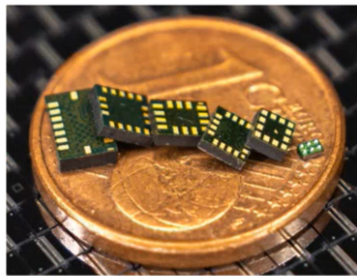

Figure 1.8. Size comparison between a MEMS sensor and 1 euro cent coin

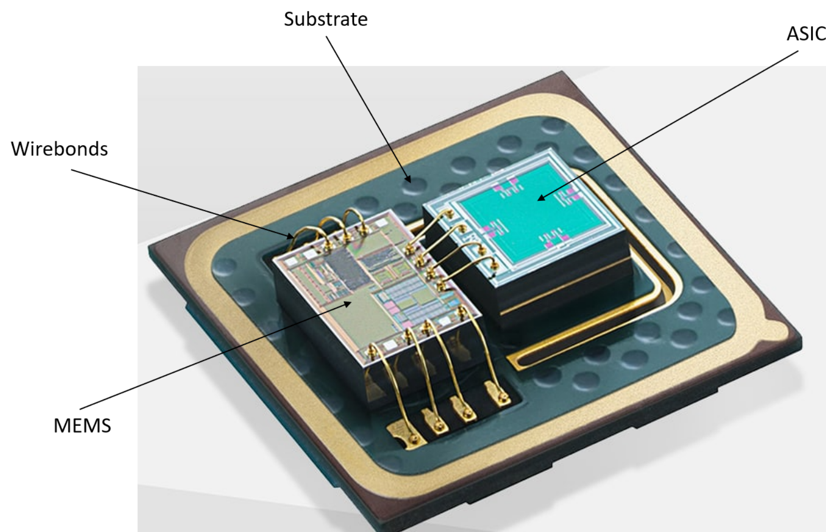

Figure 1.9. MEMS sensor module internal structure

Today, high-volume MEMS can be found in a diversity of applications across multiple markets like automotive, consumer electronics and many others. For example, modern vehicles are unthinkable without MEMS IMUs. These systems are essential for movement and impact detection, as well as position and orientation recognition.

1.5.1 Accelaration sensor

A MEMS acceleration sensor is composed of a signal processing chip (the ASIC) and a micromechanical double comb-like silicon structure (the MEMS sensor); one of the comb structures is fixed, while the other one can move and works on the principle of mass on a spring (figure 1.10).

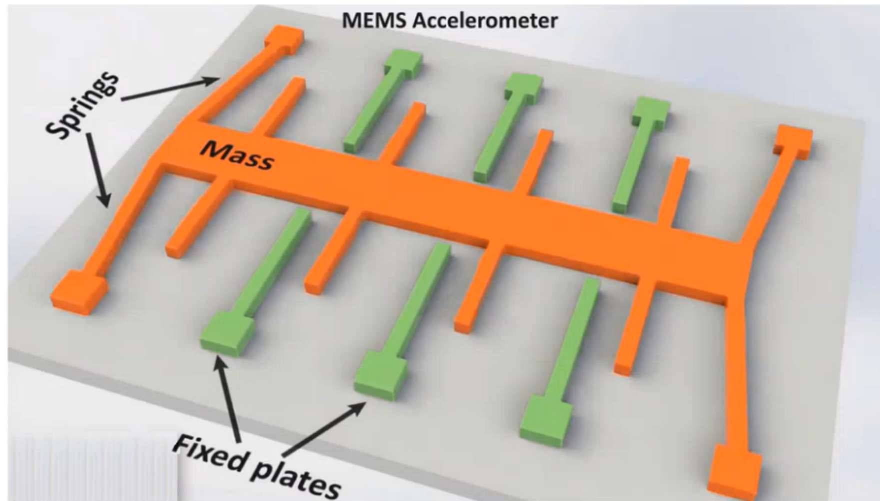

Figure 1.10. MEMS accelerometer principle for x-axis and y-axis

These two structures form what is known as *capacitive transducer*. Their fingers form a capacitor, and the distance between the microstructures determines the capacity. The moving fingers shift against the fixed fingers depending on acceleration or deceleration. As a result, the distance between them changes, influencing the capacity. The integrated electronics detect this change, convert it into a measured value and deliver it in output as a voltage signal [21].

1.5.2 Gyroscope sensor

A MEMS gyroscope sensor measures angular rotational velocity and acceleration through a vibrating structure, with the goal of maintaining a reference direction or providing stability.
One of the most common gyroscope structures is called *tuning fork*, in which a pair of test masses are driven to resonance [6].
The underlying physical principle of gyroscopes is that a vibrating mass tends to continue

vibrating in the same plane even if its support rotates. This kind of structure is subject to the Coriolis apparent force, which causes the mass to exert a force on its support; by measuring this force the rate of rotation can be determined (figure 1.11).

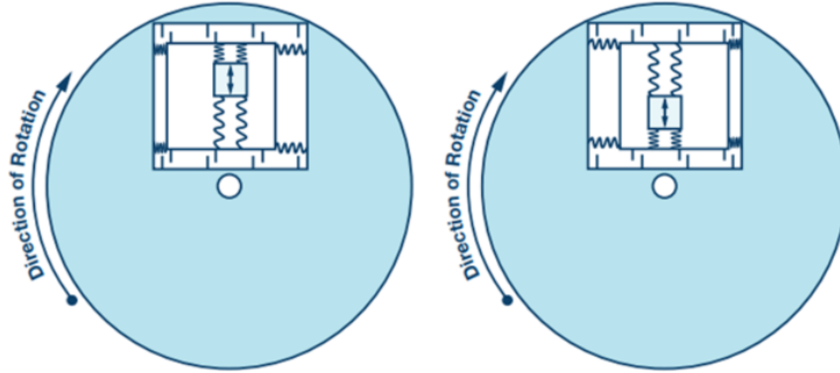

Figure 1.11. Effect of Coriolis force on a tuning fork gyroscope

To measure the Coriolis acceleration, the frame containing the resonating mass is tethered to the substrate by springs at 90° relative to the resonating motion. Like in the accelerometer, a capacitive transducer is then used to sense the displacement of the frame in response to the force exerted by the mass and output a voltage signal [38] (figure 1.12).

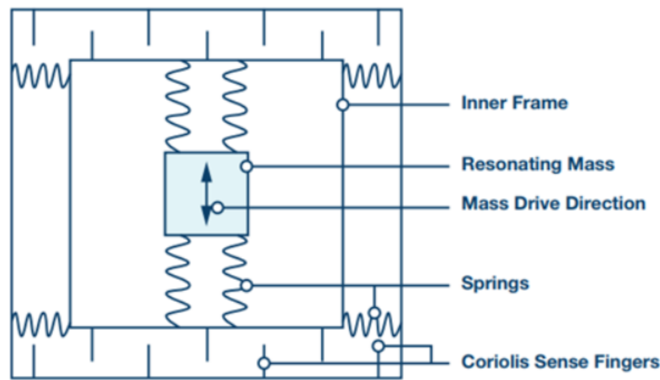

Figure 1.12. MEMS gyroscope mechanical structure

1.5.3 Bosch SMI240 Inertial Measurement Unit (IMU)

An IMU is an electronic device that measures and reports a body's specific force, angular rate, and orientation, integrating multi-axe accelerometers and gyroscopes. Typical configurations contain one accelerometer and one gyroscope per axis for each of the three principal axes.
Among the various products present in the Bosch catalog, it is possible to find the *SMI240*, a 6-in-1 IMU capable of measuring all 6 degrees of freedom: pitch (Ω_x), roll (Ω_y), yaw (Ω_z) rate and acceleration in x (a_x), y (a_y) and z (a_z).
The *SMI240* is developed for the automotive market with safety targets up to ASIL B. The target applications are especially related to hands-free and autonomous driving.
Being a safety-critical component, an onboard safety controller plus several SMs like parity bits, redundant logic, Cyclic Redundancy Checks (CRCs) and others have been added to reach the required 90% DC.
A perfect case study to understand how BST approaches safety is the CIC filter used in $\Delta - \Sigma$ Analog-to-Digital Converters (ADCs) to remove quantization errors. In figure 1.13 a CIC filter of second order (CIC^2) for consumer applications is depicted. The typical elements of this kind of filter can be observed: the integrator, the decimator and the comb, but nothing more is added when consumer electronics is the scope of the product.

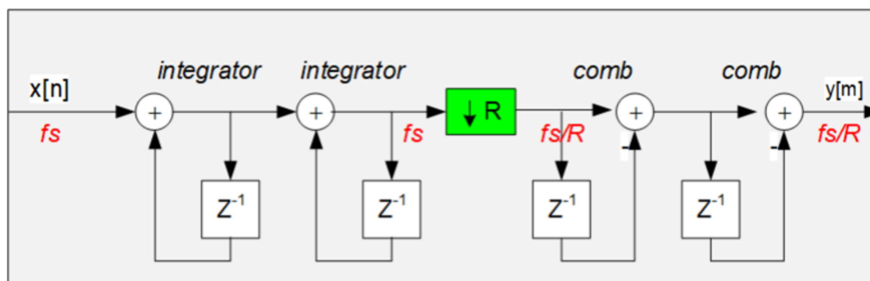

Figure 1.13. CIC^2 filter structure for consumer electronics applications

In figure 1.14 instead, a CIC^2 adapted for automotive application is represented. The underlying structure remains the same, but both combinational and sequential logic elements are protected by SMs:

- The combinational elements like adders and subtractors are doubled to implement what is called Double Modular Redundancy (DMR). Through this technique, each operation is executed by a further set of adders/subtractors, and then compared with the result obtained by the original set.
- The flip-flops are extended to support parity bits.

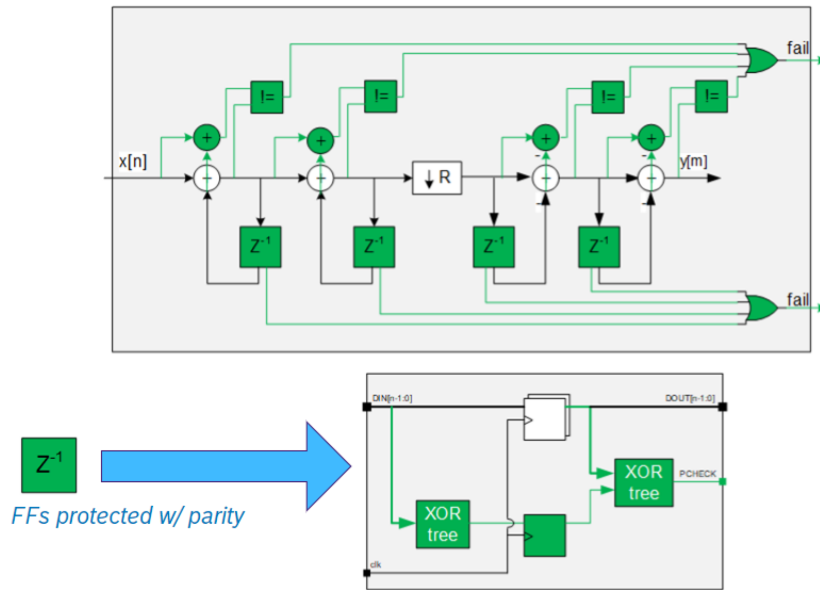

Figure 1.14. CIC^2 filter structure for automotive applications

1.6 Related works

Fault injection has been used for many years to evaluate the reliability of electronic systems and it is today a standard in the corporate environment when dealing with safety-critical components.
Depending on the technique used, fault injection environments can be classified as [27, 33]:

- Hardware-based environments. External physical sources like heavy-ion radiation [25, 22] or power supply disturbances [29, 24] are used to introduce faults directly into the hardware. Other tools instead use probes directly attached to the DUT pins to inject faults [35, 12].
- Simulation-based environments. Faults are injected into the simulated hardware. Different levels of abstraction can be used to model the DUT, from low level (VHDL, Verilog models) [34] to high level (C, SystemC models) [30]. These environments are much slower than the hardware-based ones, but they are highly controllable, flexible and customizable. On top of that, by using this technique it is possible to start the analysis process very early.
- Emulation-based environments. The target system is emulated with an Field Programmable Gate Array (FPGA) and the faults are injected in it [7].

Nowadays, one of the most common approaches used in simulation-based tools consists of expanding a pre-existing verification environment with fault injection capabilities. When dealing with this kind of environment, the biggest portion of the industry today works

with SystemVerilog and Universal Verification Methodology (UVM) which is the standard [10, 28, 16].
Anyway, some companies are still using Specman *e* for their verification purposes, but there is a huge industry gap if they want to embed fault simulation processes in their environments. In fact, the only industrial fault simulation tools based on Specman *e* known by the time this document was written are those designed by the Italian company Yogitech (now part of Intel Corporation). In the academic field instead, Specman *e* for fault injection has been analyzed in [11].
The goal of this thesis work is then to fill this gap and enable BST to easily perform fault simulations within the standard verification flow actually used in the company.

Chapter 2

Fault Injection e Verification Component (FIeVC)

A fault simulator is a software tool that computes the behavior of a circuit in the presence of faults to help users develop robust diagnostic tests and verify that SMs meets the fault injection testing requirements.
Fault simulations can be executed using different algorithms [1]. The *FIeVC* is based on a *parallel* algorithm, which offers a good balance between the low complexity of a *serial* algorithm and the high performance of a *concurrent* algorithm. More complex techniques, like the *deductive* and the *differential*, have not been considered to avoid getting lost in the theory behind them.
The main strengths of the *FIeVC* are:

- Non-invasive environment. Despite being built on a pre-existing verification environment, the *FIeVC* is completely transparent to it. In fact, it only acts on the DUT, leaving the underlying verification environment untouched.

- Ease of use. It is possible to choose between the fault simulation flow and the functional verification flow by simply defining or not the *BST_FAULT_SIM* environment variable. Thanks to the *ifdef* directive it is possible to check whether *BST_FAULT_SIM* is defined or not and compile the testbench and the required files accordingly.

- Adaptability. The *FIeVC* can handle from basic to very complex designs at all hierarchy levels. Thanks to the extensive use of REs and BST naming conventions, the files composing the *FIeVC* are automatically readapted to the current DUT before the elaboration phase with no particular effort required from the user.

- Flexibility. The *FIeVC* can be extended or modified at will. Thanks to the isolation of tasks in different files, it is possible for example to extend the analysis to new fault models by simply modifying one file.

2.1 Generic parallel fault simulator structure

For the sake of simplicity, the *FIeVC* simulates in parallel only two circuits, the golden and the faulty one. Nevertheless, with some small changes in the preparatory scripts (section 2.2) and in the environment itself (section 2.3) it is possible to increase the degree of parallelism and in turn the performance.
The general structure of a parallel fault simulator is depicted in the figure below (figure 2.1). The original BST verification flow is highlighted in green, while the additional components required for the fault simulation are highlighted in red.

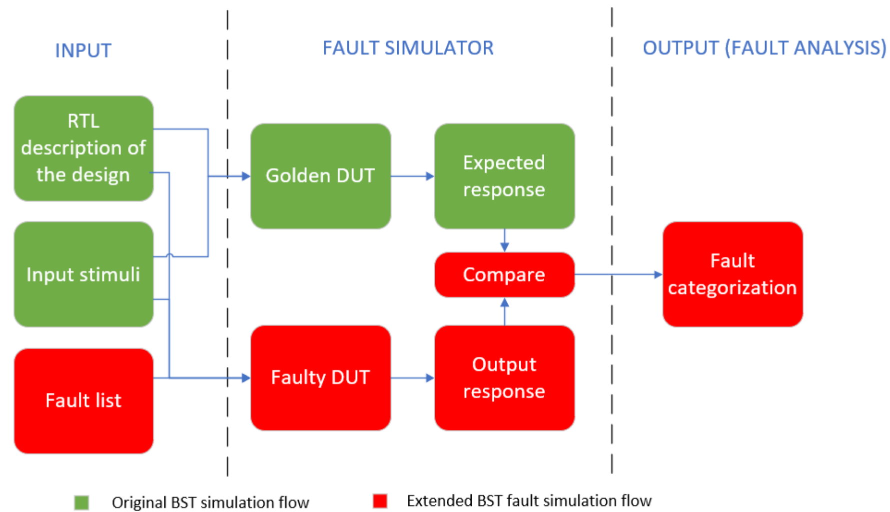

Figure 2.1. Parallel fault simulator diagram

The additional elements required to perform a fault simulation are then:

- A fault list. It is a file containing all the injectable faults of the design.
- An additional DUT. To ensure accurate reference results, the fault-free golden simulation must continuously run in parallel with the faulty simulation; this means that a single DUT is not enough and the TB must be readapted for this purpose.
- Auxiliary signals. These signals ease the comparison process. Their usage makes the comparison of outputs directly possible inside the TB indeed. The benefits of this solution are primarily related to the possibility of keeping the *FIeVC* unaware of the actual DUT, bringing an overall increase in generality and flexibility. More details on these signals are given in section 2.2.2.
- A fault injection and fault analysis tool. This is trivial and it is of course the task of the *FIeVC* itself.

2.2 Preparatory scripts

As seen in section 2.1, some preliminary operations are needed to make the fault simulation possible. Anyway, implementing these features manually would be exhausting and, more importantly, would completely hinder the flexibility and ease of use of the tool.
Thus, the fault list generation and the TB readaptation are completely entrusted to a couple of scripts: a Tool Command Language (TCL) script for the fault list generation, and a Python script for the TB readaptation.

2.2.1 Fault list generation script

The goal of this script is to generate a text file containing one fault per row.
Canonically a fault item is characterized by three parameters: fault site, fault kind and injection time (section 2.3.1). However, this script simply outputs a list of injectable fault sites instead of full fault items. In this way, both the fault kind and the injection time can be managed directly by the *FIeVC*; whenever a fault site is picked from the list, it is the *FIeVC* that decides which specific fault (SA0, SA1, etc.) to inject and when to inject it based on the settings defined by the user in the Sequencer (section 2.3.5) and the BFM (section 2.3.6).
For this script, TCL is preferred over Python because it can be executed as a pre-simulation script directly through the Xcelium™ Simulator TCL Command-Line Interface (CLI). With this setup, TCL commands are available while having full access to the elaborated design. It is therefore possible to retrieve all the input ports, output ports and internal nets of the design automatically using the *find* command.
The steps executed by this script are the following:

- Asks the user for the scope of the injection, supporting multiple components and recursive injection.
- Issues the TCL *find* command on these components to retrieve input ports, output ports and internal signals.
- Splits the vector signals into single-bit signals to make them injectable.
- Changes the path of the signal from ":dut:*" to ":dut_faulty:*" so that the faulty DUT and not the golden one is injected.
- Lists all the obtained fault sites, one per row, in a text file.

2.2.2 TB readaptation script

This script takes care of everything related to the TestBench (TB) readaptation in order to have an environment that accepts fault injection and fault analysis. The goal is therefore to duplicate the DUT for the fault injection task and to add the comparison logic for the fault analysis task.
Given that this script directly modifies the files of both the TB and the *FIeVC*, it must be executed prior to the elaboration phase so that the environment used during the simulation is up-to-date.
For what concerns the first task, the DUT duplication, a complete parsing of both the VHDL design and the DUT instantiation in the TB is required to retrieve everything needed to perform it (input/output signals, generic/port mapping and signal assignment). With all these data at its disposal, the script can now declare the second DUT changing its name from "dut" to "dut_faulty". On top of that, it renames all the outputs from "*" to "*_faulty". The reason behind this renaming is very straightforward; two components with the same output signal would generate a multiple-driven net error. Finally, also input names are modified from "*" to "*_buf". The reason behind this second renaming is much more subtle, and it is better explained later in this document (section 2.6.1).
For what concerns the second task instead, the comparison logic implementation, two choices are possible:

- Bring the outputs belonging to teh DUT and DUT faulty in the *FIeVC* and implement the comparison logic there.
- Implement the comparison logic directly inside the TB.

The first is probably the more canonical of the two. In fact, this solution ensures that the *FIeVC* is the only actor handling the fault simulation process. Nevertheless, it comes at a higher cost: having an unknown number of outputs means that comparing all the golden outputs with all the faulty outputs requires a for loop that cycles through and compares them one by one at each and every clock cycle. Therefore this solution has a major impact on the total simulation time.
The second solution instead brings a portion of the fault simulation process outside the *FIeVC*, but it guarantees more flexibility and also comes at zero cost considering that the TB is already under a reshaping process. For these reasons, this solution is the implemented one.
However, before moving on with the comparison logic explanation, a short description of the structure of a fault-tolerant block is crucial. As shown in figure 2.2, these blocks are composed of a module and a SM connected to it. The module is the functional portion of the design, which implements the desired functionality, while the SM is of course the safety portion, which checks for possible errors inside the module. Consequently, also the outputs of a fault-tolerant block are split into functional outputs and safety outputs.
Fortunately, in BST designs safety outputs can be distinguished from functional outputs by the presence of the "_fail" string at the end of the signal name.
With this in mind, the new signals needed for the comparison and analysis process are:

- One "*_xor" signal per output. These signals take the value of "*" XOR "*_faulty" and are used to know where the error has been propagated.

- A "mismatch_detected" signal. This signal produces the OR of all the above-mentioned "*_xor" signals, meaning that it assumes the value '1' whenever there is a mismatch in at least one functional output (figure 2.3). Before the OR with other signals, vector "*_xor" signals must be squashed into a single bit by performing a reduced OR.

- A "fail_detected" signal. Similar to the "mismatch_detected" but for the safety outputs (figure 2.4). This signal notifies that the SM detected an error and it is therefore used to trigger for the fault analysis process.

Finally, the last operation performed by the script is to populate the *FIeVC* Signal Map (section 2.3.2) to have direct access to all these signals from the *FIeVC* too.

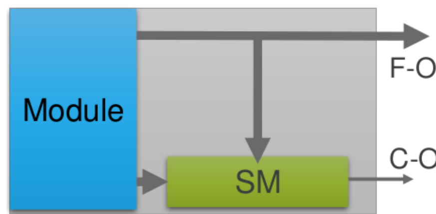

Figure 2.2. Functional unit representation (credits to Cadence®)

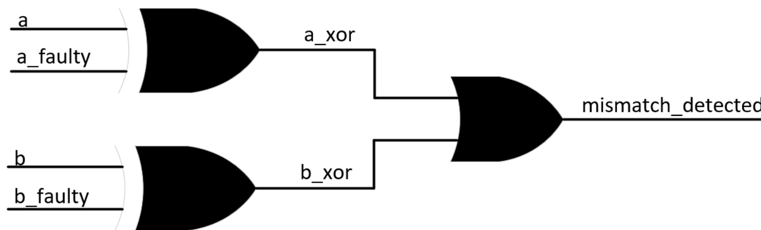

Figure 2.3. Simplified representation of the "mismatch_detected" logic

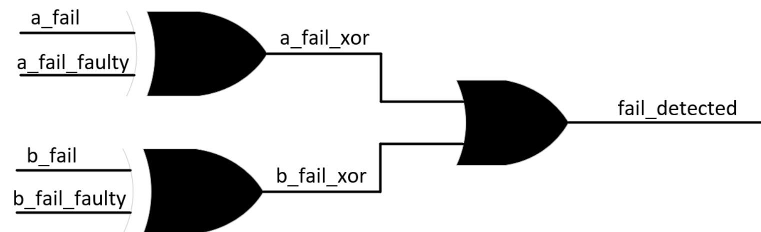

Figure 2.4. Simplified representation of the "fail_detected" logic

2.3 *FIeVC* structure

With the setup required for the fault simulation ready, it is now time to see how the fault injection and fault analysis are implemented.
A correct implementation of these two concepts is indispensable for a successful fault simulation and it is in fact the problem addressed by the core of this thesis, which is the *FIeVC*.
In figure 2.5, the structure of the *FIeVC* and how it is connected with the TB is shown. The upper part of the figure shows the modified TB while, in the lower part, it is possible to notice all the inner components of the *FIeVC* which are: *Signal Map, Monitor, Analyzer, Sequencer and BFM.* These components are all declared inside a further component called *Env*, which acts as a wrapper and also binds input ports to output ports to establish the required connections.

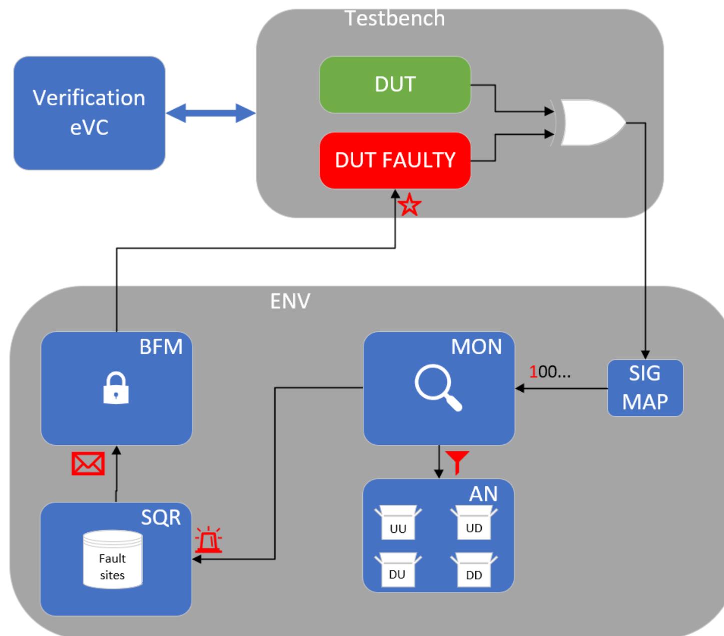

Figure 2.5. *FIeVC* structure with simplified connections

The connections between the above-mentioned components are performed through the so-called *e* ports, which in Specman *e* are the equivalent of the more common TLM ports. *e* ports enhance the portability and interoperability of the *FIeVC* by making the separation with the DUT possible.
There are two ways to use *e* ports:

- By creating connections between the simulated object (everything present inside the TB), and the *FIeVC* components. These ports are called *external.*
- By creating connections between two *FIeVC* components. These ports are called

internal or *e2e*.

For what concerns the ports kind instead, the ports used are mainly of two kinds:

- *Simple* ports. Used to directly access and transfer data.
- *Event* ports. Used to transfer events between the units for triggering or synchronization purposes.

In the next sections, all the *FIeVC* subcomponents are explained in detail.

2.3.1 Fault item

To easily transfer information about the current injected fault and the next one, it is necessary to define an atomic description that represents the fault and can be accepted by the *e* ports.
A fault item is therefore used to model the physical fault. This fault item is composed of three parameters [2]:

- Fault site. It indicates the net on which the injection must be performed. The fault site is assigned to the fault item by the Sequencer (section 2.3.5), randomly extracting one fault site from the fault site lists.
- Fault kind. It indicates the category of fault model injected. It could be a SA0, a SA1, a SEU, etc.. The fault kind too is assigned to the fault item by the Sequencer (section 2.3.5), randomly choosing between the fault kinds declared by the user.
- Injection time. It is the time in which the error should rise and manifest itself inside the circuit. Having taken into account only SAFs in this version of the *FIeVC*, the injection time is automatically set to an instant of time between the end of reset and the start of test operations.

2.3.2 Signal map

This component is automatically generated by the TB readaptation script (section 2.2.2) based on the auxiliary signals included in the TB. It acts as a connector between the TB and the *FIeVC*, having direct access to all the "*_xor", the "mismatch_detected" and the "fail_detected" signals using simple ports. By binding a simple port to the full HDL path of the desired signal, it will continuously reflect the value of that signal making it accessible from the *FIeVC*.

2.3.3 Monitor

The Monitor is in charge of keeping track of all the changes in the comparison logic lines to detect possible mismatches and consequently perform the necessary operations.
Three functions are executed inside this component:

- *generate_fault()*. Whenever a new injection can be performed, the Monitor notifies the Sequencer through an event port, called *generate_next_fault*.

- *monitor()*. The simulation of the current injected fault lasts until the test reaches its end, or until a mismatch in some safety outputs is observed through the *fail_detected* line. When one of these two situations occurs, it is possible to proceed with fault classification in the Analyzer (section 2.3.4), and the next injection.
When the Monitor executes this function it waits for one of these two events to trigger the analysis and proceed with the next injection. In the meantime, it also checks if any mismatch happens on the *mismatch_detected* line because the Analyzer requires this information to correctly classify a fault.

- *update_tlm_ports()*. The simulation time at which the first mismatch and the first fail arise are saved too, and passed to the Analyzer for classification purposes. This transfer is performed using *tlm_analysis* ports called *mismatch_time_port* and *fail_time_port*. This kind of port is similar to an *e* port but also supports multiple connections and broadcasting.

2.3.4 Analyzer

Whenever the simulation of a fault item is over (because the fault was detected or because the test ended) it is possible to proceed with its classification, depending on the obtained outcome.
For later comparisons, in the Analyzer the same fault classification used by XFS is implemented (figure 2.6).

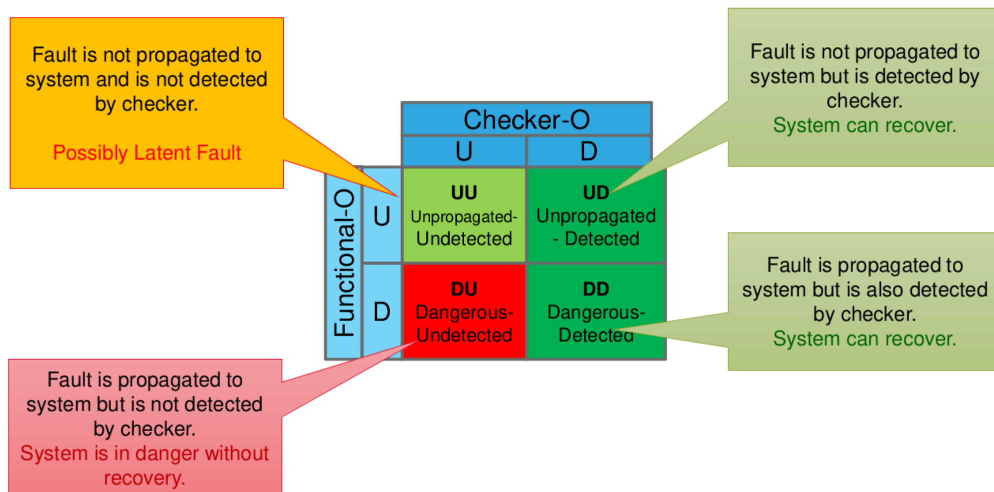

Figure 2.6. *FIeVC* fault classification (credits to Cadence®)

Faults are thus divided into four categories:

- Unpropagated Undetected (UU). The fault is neither propagated to a functional output nor detected by a safety output. In this case, it is not possible to jump to firm conclusions, but there are chances that the fault could be a safe fault.

- Unpropagated Detected (UD). The fault is not propagated to a functional output but detected by a safety output. In this case, the system is able to recover.
- Dangerous Undetected (DU). The fault is propagated to a functional output but not detected by a safety output. In this case, the system is in danger because it means that it is not possible to recover from this error.
- Dangerous Detected (DD). The fault is both propagated to a functional output and detected by a safety output. In this case, the system is able to recover too.

Three functions are executed in this component:

- *init_report()*. The first thing to do for a good and complete analysis is to generate a report file that can hold the results of all the injections.
  The file is composed of four fields: *fault site, fault kind, fault category, FRTI*. This function creates the new file and initializes it with the four fields division.
- *analyze_fault()*. As soon as the analysis is triggered, the value of *mismatch_time* and *fail_time* are read from the two *tlm_analysis* ports. With these two numbers, it is possible to check if, and at which instant of time the fault was propagated to a functional or safety output and proceed with the classification (figure 2.7). Saving also the difference between *fail_time* and *mismatch_time* (the FRTI) allows the user to check if any SM has violated its timing constraints.
- *generate_summary()*. When the fault simulation campaign is over, the overall number of obtained UU, UD, DU and DD is written at the end of the report file.

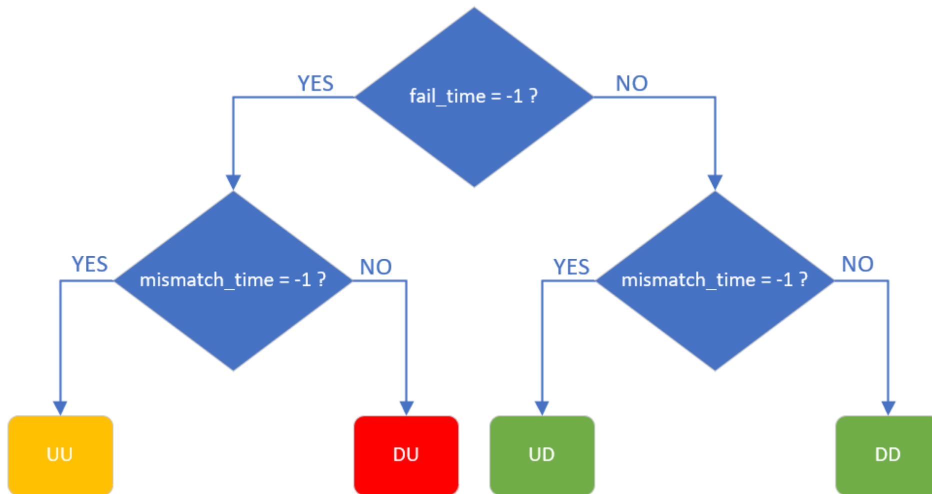

Figure 2.7. *FIeVC* fault classification flow

2.3.5 Sequencer

The sequencer works as a fault item generator. It is connected to the Monitor through the *generate_next_fault* port to receive a notification whenever a new fault item is needed.
The fault simulation, if not stopped by the user, will proceed until all possible faults have been injected. When this happens, another event port, called *fault_list_over*, is used to trigger the end-of-simulation phase.
Two functions are executed inside this component:

- *fill_fault_sites_list()*. At the very beginning of the simulation, the Sequencer reads the fault sites list generated with the fault list generation script (section 2.2.1) and creates some internal copies using lists of strings, more precisely one for each fault kind. Despite slowing down the start-up process to make these copies, this solution saves a lot of time in the long run. The main advantage is that whenever a new fault item must be created, there is no need to access and reread the file. In this way, the accesses to the file are reduced and in turn the simulation time.

- *pick_fault()*. With the setup obtained through the previous function, creating a fault item basically means picking a fault site from one of the lists, and assigning the corresponding fault kind and injection time. The execution of this function is triggered by the *generate_next_fault* port.

2.3.6 Bus Functional Model (BFM)

The BFM acts as the fault injector of the system, and for this reason, it is also called Driver.
Two functions are executed inside this component:

- *force_fault*. As soon as the simulation starts, and whenever the analysis phase of an injected fault is over, the BFM retrieves from the Sequencer a new fault and injects it on the DUT by executing on the simulator the command "*force (fault_item.fault_site) (fault_item.fault_kind)*".

- *release_fault*. Before a new injection cycle is allowed to start, the BFM must release the old fault by executing on the simulator the command "*release (fault_item.fault_site)*".

2.4 Simulation flow

With the splitting of all the different tasks between these components, the simulation flow turns out to be very linear and manageable. It is divided into three phases: init phase, core phase and end phase. In each of the three phases, one or more functions per component are executed. What is astonishing is that the complexity behind all these functions is completely hidden from the user who can launch the simulation by simply using the *xrun* command with the desired flags. The complete simulation flow is depicted in figure 2.8.

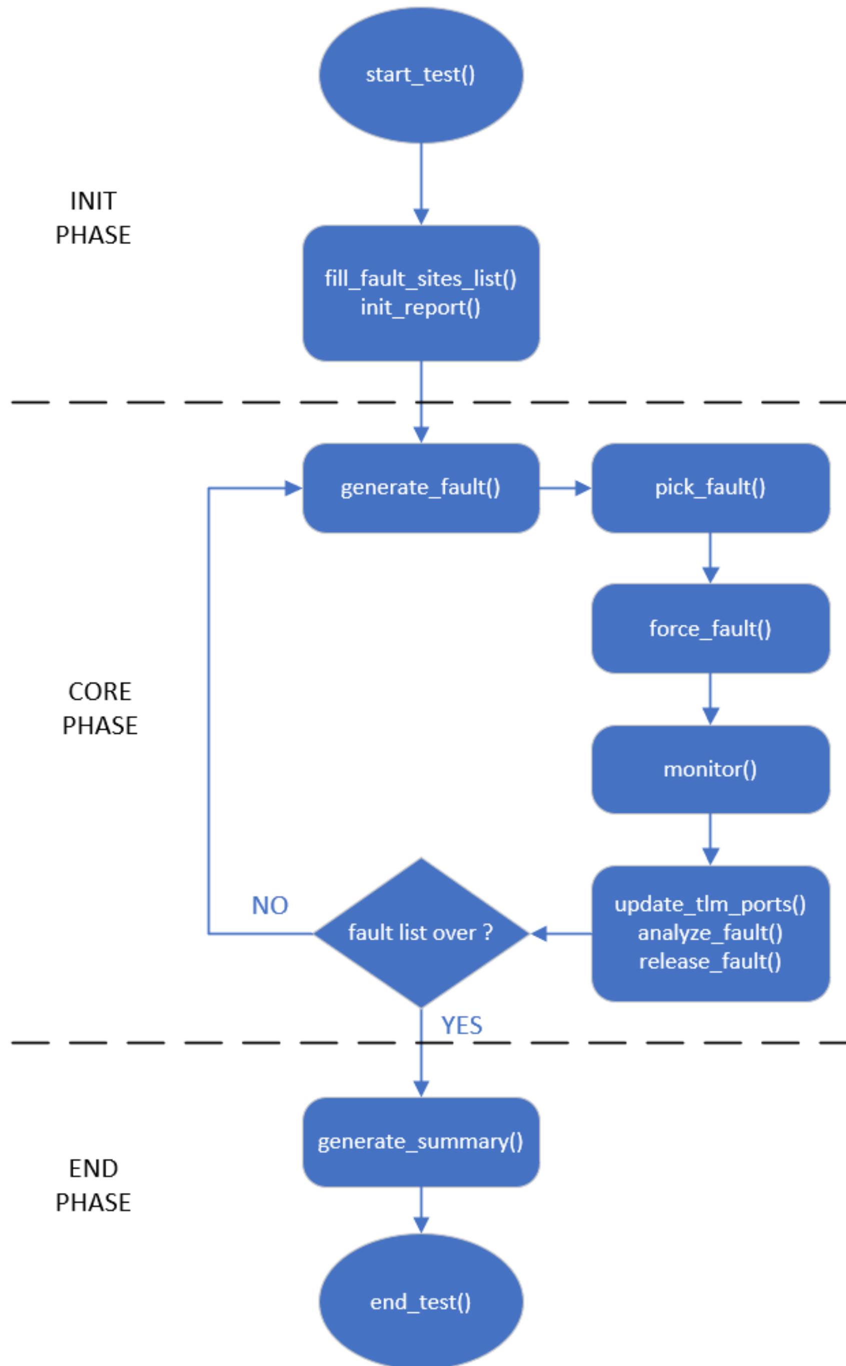

Figure 2.8. *FIeVC* fault simulation flow

2.5 Resetting the DUT using the Testflow feature

The flow described so far does not take into account a simple yet crucial limitation: whenever the BFM performs a new injection, it is required to reset the DUT and restart the test from the beginning.
Resetting the DUT is needed because most probably the previous error spread inside the circuit corrupting sequential elements, so the DUT must be restored to a known and safe state. The first solution that comes to mind is very simple: close the simulation and restart it, injecting the new fault as soon as the new simulation starts.
Although it sounds like an easy problem to solve, implementing this first solution brings a high overhead in terms of simulation time mainly due to the time required by the Xcelium^{TM} Simulator to open and retrieve the design snapshot. On top of that, the fault lists generated in the Sequencer would be useless because they would get overwritten at every new launch of the simulation, making it impossible to keep track of the already injected faults. This means that rolling back to a single fault list contained in a file would be required, further incrementing the simulation time. Considering all the downsides of this solution, some time was spent to give a look at all the tools and features that Specman *e* has to offer to check if another, less consuming solution could be implemented.
The solution found relies upon the advanced Testflow feature. The benefits of using this approach are multiple and, if considered altogether, they heavily increase the performance of the *FIeVC*.
By making the *FIeVC* components and the test itself "Testflow aware", it is possible to partition the simulation and test execution into multiple subphases. With this structure, the simulator guarantees that a unit proceeds to the next phase only after all activities related to the current phase are done. The simulation therefore shifts from a monolithic execution of the test to a multi-threaded version.
There are 8 predefined phases, 6 of which are used in the *FIeVC*. The implemented phases are:

- ENV_SETUP. An objection is raised to make the test start.
- RESET. Both DUTs are reset to accept a new injection.
- INIT_DUT. After the reset but before the test execution, the faulty DUT is initialized with a new injected fault.
- MAIN_TEST. With the faulty DUT ready, it is possible to start the test sequence.
- FINISH_TEST. Whenever the test sequence ends or a mismatch in a safety output is detected, the test sequence is halted. From here the fault is analyzed and then the test is brought back to the restart phase.
- POST_TEST. When the fault list is over, instead of going back to the reset phase the test proceeds with the objection dropping and its natural conclusion.

The biggest strength of this approach is that it is possible to restart the test from a specific phase at run time regardless of which phase is currently executed.

By exploiting the phases, the built-in synchronization feature and the rerun functionality, it is possible to reset the DUT over and over again whenever a new injection must be performed.
The functions seen in section 2.3 now become threads, each executed in a specific Testflow phase as shown in table 2.1.

Phase \ Component	BFM	Monitor	Analyzer	Test
ENV_SETUP				start_test()
RESET				drive_reset()
INIT_DUT	force_fault()	generate_fault()		
MAIN_TEST		monitor()		drive_sequence()
FINISH_TEST	release_fault()	rerun(RESET)	analyze_fault()	
POST_TEST				stop_test()

Table 2.1. Thread partitioning among the different Testflow phases

2.5.1 Simulation flow using the Testflow feature

The flow with this new functionality is very similar to the one explained in section 2.4 but more focused on the execution of Tesflow phases rather than on the single functions, as shown in figure 2.9.

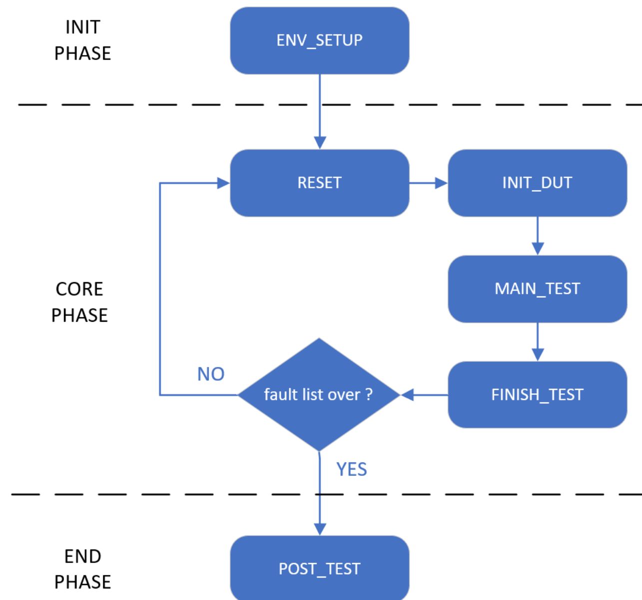

Figure 2.9. *FIeVC* fault simulation flow with active Testflow feature

2.6 Backpropagation issue

Unfortunately, a strange behavior was observed when injecting some faults during the initial tests performed on the *SMI240*. More precisely, it looked like some faults were observed on certain safety outputs not belonging to the forward cone of logic of the injected node. The presence of a problem in the environment was made clear by the detection of the fault effects even on the golden DUT, causing the entire simulation to fail. To make the *FIeVC* usable was mandatory to understand the source of this problem and solve it.
To determine the causes of this behavior, a manual inspection of the signals' propagation on the schematic tracer was performed. The response of the circuit after an injection is shown in figure 2.10. As observed from this picture, forcing a signal to assume a value also forces all other signals connected to it to assume the same value, no matter if it belongs to the forward or backward cone of logic.
The problem thus arises because the Xcelium^{TM} Simulator *force* command does not have protection against backward propagation of the injected fault which is therefore propagated throughout the whole net, up to the driver. This means injecting a signal is always equivalent to injecting its driver and the whole net.
Considering that the VHDL port mapping connects two signals without interposing anything between them, very likely the design will have big nets composed of different signals. Depending on where the driver is located, the backpropagation is classified as external or internal, and both of them require countermeasures to stop the propagation.

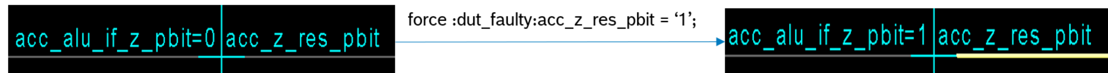

Figure 2.10. *force* command propagation error

2.6.1 External backpropagation

Among the two, external backpropagation is the one that does not allow performing fault simulation at all. This means that a solution must be found no matter what.
This type of backpropagation is observed when the injected node is directly connected to a DUT faulty input. If this is the case, it means that the driver of that node is located outside the DUT faulty, and drives both the faulty and the golden DUT. The combination of this structure and the *force* command behavior explained before causes the fault to propagate from the faulty to the golden DUT because injected on the driver's output which is external.
The solution to this problem is to interpose a buffer between all the DUT faulty input lines and their corresponding original drivers (figure 2.11) so that the input signals' drivers become the buffers themselves, reducing the scope of injection merely to the faulty DUT.
The buffer insertion is performed through the TB readaptation script, as explained in section 2.2.2.

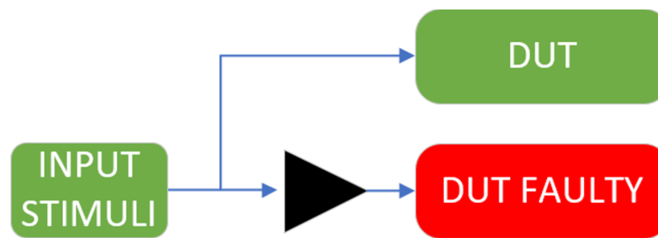

Figure 2.11. Extended TB structure to solve external backpropagation issue

2.6.2 Internal backpropagation

Unfortunately, this problem does not show only at the DUT faulty boundary but also inside the component for the same reason. This means that faults cannot be injected on component/gate inputs, but only on outputs (because they act as drivers for whole nets). This is certainly a limitation of the *FIeVC* that must be considered when performing a fault campaign.
To temporarily patch this issue, the fault list generation script (2.2.1) is modified so that the list contains not the signals themselves but rather their drivers. Before proceeding with the simulation, the generated list of drivers is re-elaborated to remove all the duplicates. This solution makes it possible to perform the entire fault campaign and compare the results with XFS with the downside of reducing the scope of the fault injection to only a portion of the injectable nodes.
Two possible solutions for this problem are here reported but not implemented in the final version of this thesis:

- Using fast recompilation feature of the Xcelium™ Simulator to perform dynamic buffer insertion. The fast recompilation allows recompiling a single VHDL file of the entire design, reducing the time needed for subsequent compilations. In this way, it is possible to add a buffer where needed, recompile the file and execute the simulation. Anyway, this solution is not compatible with the Testflow feature, thus bringing a huge loss in terms of performance.
- Creating a new Python script that duplicates every single file in the design with a new version that puts buffers on all the components.

Chapter 3

Xcelium Fault Simulator (XFS)

3.1 Main features

XFS is an easy and quick-to-setup tool able to operate within the existing Xcelium^{TM} Simulator compiled engine, allowing for the seamless reuse of functional verification environments. This means that it is possible to set up and perform a complete fault simulation campaign by simply extending the simulation flow, without any required modification on the pre-existing environment.
Contrary to the *FIeVC*, XFS supports only the serial fault simulation algorithm when dealing with VHDL components. On top of that, this tool does not have a fast reset feature like the one offered by the *FIeVC* with the Testflow feature, meaning that the only way to perform multiple injections in sequence is to close the running simulation, restore the elaborated snapshot and relaunch the simulation from scratch. Later in section 4 the impact that these two missing features have on the overall performance is shown.
Aside from all the files belonging to the original verification environment, two new files are needed to use XFS:

- A fault specification file. Through this file, some preliminary information about the fault simulation is passed to XFS. In particular, the command *fault_target* is used, followed by the path of the component or sub-component on which the fault injection is executed. This command also accepts two flags: *-type* to specify the type of fault to inject (SA0, SA1, SET, SEU or all of them), *-select* to specify which types of nets are going to be injected (ports, cells or sequential signals only).

- A strobe file. This file is used to define the splitting between functional and checker outputs as discussed in section 2.2.2. To increase the reusability of this flow, instead of generating a strobe file for each DUT, the tool uses a further TCL script that automatically sets both functional and checker outputs by analyzing the component's output ports during the elaboration phase. Lastly, this script is also used to specify

strobing events for the signals. When the faulty DUT is simulated and these events are triggered, the current value of all signals is compared with those expected.

3.2 Simulation flow

The overall simulation flow is quite different from the one used in the *FIeVC*, especially because of the different algorithm used.
Using a serial algorithm means that it is possible to simulate only one DUT at a time. This brings the need to perform a preliminary golden simulation in which a faultless DUT is simulated. This simulation is performed using the classic *xrun* command. By using the strobe events defined in the strobe file (section 3.1), the expected behavior of the golden DUT is extracted and saved in the so-called fault database.
With the database ready, the desired fault list is generated by launching the *xfsg* command. It is now possible to start all the simulations one after the other comparing at each and every occurrence of the strobe events the obtained behavior with the expected behavior stored in the database. The fault classification is in turn stored inside the database, and at the end of the fault simulation campaign, a report can be generated through the *xfr* command.
The scheme depicted in figure 3.1 summarizes all the steps mentioned above to get a working fault simulation.

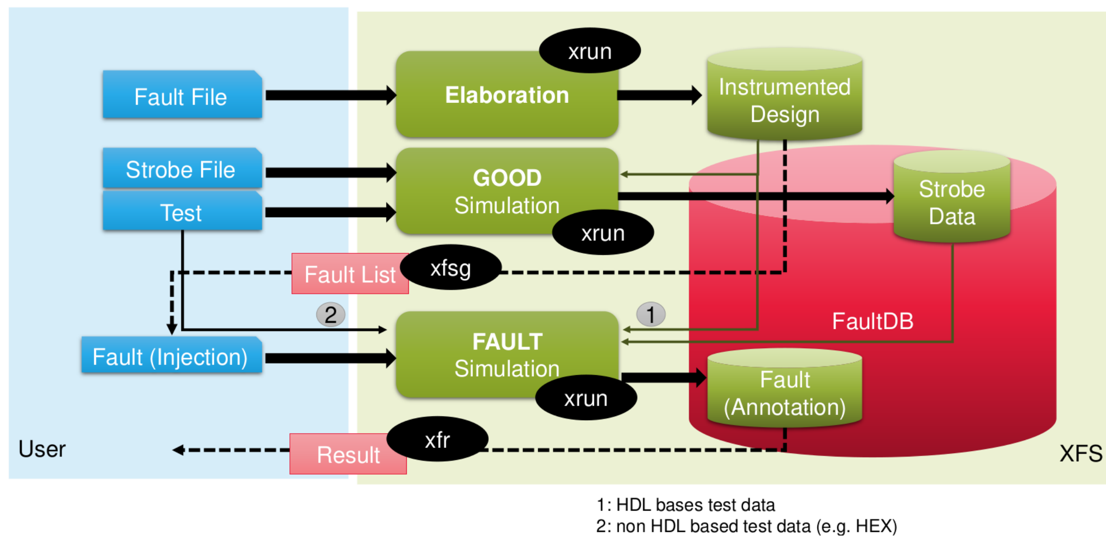

Figure 3.1. XFS fault simulation flow (credits to Cadence®)

Regarding fault categorization, the same approach described in section 2.3.4 is used.
The usage of a Makefile is highly suggested to wrap all the required commands and relative flags and options under a single Make entry. The great advantage that Makefile also offers is the possibility to declare environment variables whenever launching a new command. Thanks to the Makefile, it is no longer necessary to split and rewrite all the files for the

functional verification, the *FIeVC* fault simulation and the XFS fault simulation. Simply wrapping the non-shared portions of code around an *ifdef* directive makes it possible to execute only those portions of code required by the desired procedure.

Chapter 4

Results

To validate the *FIeVC* performance a small portion of the *SMI240* is used as DUT. In particular, the chosen section is the accelerometer datapath. A sample of 1,000 faults (500 SA0 and 500 SA1) is extracted from the total pool of 37,542 faults of this component, such that each fault site is a driver to avoid different behaviors when simulating with XFS and with the *FIeVC* (see section 2.6.2). At the end of both simulations, a report with all the necessary information is generated.
In figure 4.1, the comparison between the classification done by the two tools is shown. As can be seen, XFS is able to classify only 840 faults out of 1,000, most probably because it is not able to inject errors in those sites. By evenly splitting these 160 not-injected faults among the other four categories, UU and DU have a pretty much perfect match between the two tools. The only negative note of the *FIeVC* is a strange behavior observed in the classification of DD and UD. The tool classifies in fact a lot of expected DD as UD. The causes of this tendency toward the UD classification over the DD should be probably sought again in the effect that the *force* command has on the system (see section 2.6).

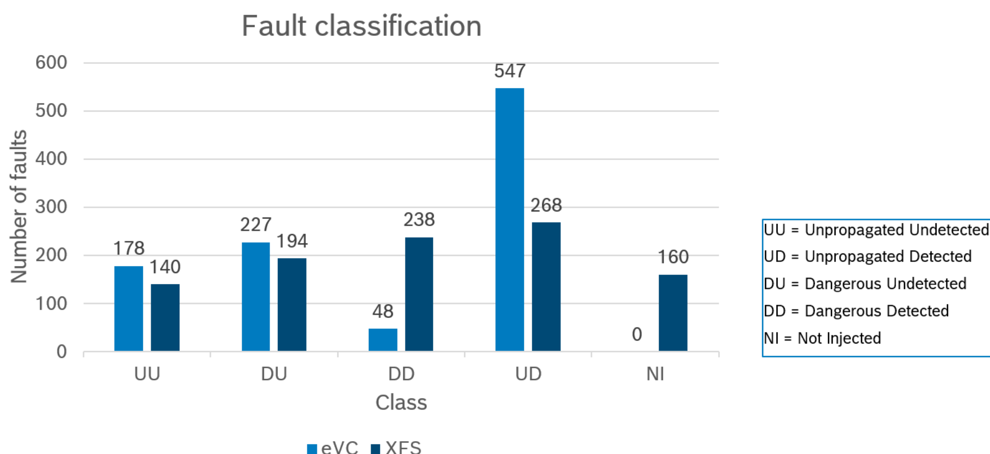

Figure 4.1. Comparison between XFS and *FIeVC* fault classifications

The Key Performance Indicator (KPI) used to check the consistency of the *FIeVC* w.r.t. XFS is the TC. The TC is in fact defined as the percentage of detected faults out of the detectable faults in the design. In other words, the TC is the ratio between the number of detected faults over the injected faults. This means that the global behavior of the two tools can be considered similar if their TCs are similar.
As expected, the two TCs are the same and in figure 4.2 the trivial comparison between them is shown. Much more relevant in this picture is the simulation time instead. As shown in the figure, in fact, the *FIeVC* simulation only takes 7 hours, against the 22 hours taken by XFS. This means that the *FIeVC* takes 68% less time to perform a full simulation, which is a huge improvement considering that full fault simulations can take up to some weeks or even months. The two main actors that bring to this outstanding performance are the run-time Testflow reset and the parallel algorithm, as already stated earlier.

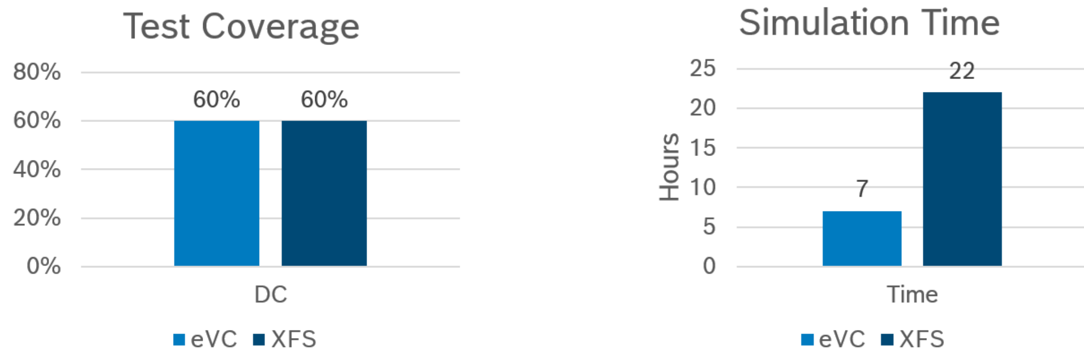

Figure 4.2. Comparison between XFS and *FIeVC* TC and total simulation time

Chapter 5

Conclusions

This thesis is focused on the development of a fault injection and fault analysis tool for safety-critical applications SoCs, where the impact of faults can be potentially disruptive. The environment is developed in Specman *e* language to be fully compatible with the underlying verification environment and with the goal of being non-invasive, easy to use, highly adaptable and flexible.
The design of this tool is based on the most common fault simulator architectures, in which the main components are:

- The driver to inject new faults.
- The monitor to check if faults propagate to outputs.
- The analyzer to classify each fault.
- The fault database.

Comparative results with the well-known XFS tool show that the usage of the parallel algorithm over the serial one, and the implementation of the optimized Testflow reset procedure cuts the total simulation time by 68% while keeping the test coverage level unchanged. This means that even if there is not a perfect equivalence in the fault classification between the two tools, the overall analysis brings the same result. On top of that, an improvement of three times in terms of performance is crucial in the fault simulation field, where simulations and analyses can last up to several days (if not entire months).
The main limitation of the architecture at the moment is the impossibility of correctly injecting all the existing fault sites due to the problem of internal backpropagation presented in section 2.6.2. Nevertheless, this limitation could be removed in the near future by finding a way to stop backpropagation.
The future of the *FIeVC* in BST seems bright, but still, a lot of improvements can be made to make it more fungible and part of the standard BST verification process. In the future, the following points could be addressed to unleash the full potential of this tool:

- Solve the backpropagation issue.
- Evaluate the benefits of parallel fault injection with multiple faulty DUTs running in parallel to optimize simulation time.
- Expand the environment to make more complex and more precise analyses.
- Perform new tests on bigger designs like system-level architectures, or more complex ones like gate-level components to check if the tool is adaptable to all these different scenarios.
- Start a validation process with the ISO to ensure the level of compliance of this tool and check whether it can be used on real application projects for external customers too.

Bibliography

[1] Algorithms for Fault Simulation. URL http://ece-research.unm.edu/jimp/vlsi_test/slides/html/fault_simulation1.html. Accessed: 2023-07-17.

[2] Fault Injection. URL https://en.wikipedia.org/wiki/Fault_injection. Accessed: 2023-07-19.

[3] Determining Diagnostic Coverage. URL https://www.kvausa.com/determining-diagnostic-coverage/. Accessed: 2023-07-14.

[4] ASIL Certification for HW Components and HW Evaluation, 2021. URL https://www.functionalsafetyfirst.com/2021/08/asil-certification-for-hw-components.html. Accessed: 2023-07-13.

[5] Automotive Safety Integrity Level, 2023. URL https://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level. Accessed: 2023-07-13.

[6] Vibrating structure gyroscope, 2023. URL https://en.wikipedia.org/wiki/Vibrating_structure_gyroscope. Accessed: 2023-07-13.

[7] Zain Ul Abideen and Muhammad Rashid. Efic-me: A fast emulation based fault injection control and monitoring enhancement. *IEEE Access*, 8:207705–207716, 2020. doi: 10.1109/ACCESS.2020.3038198.

[8] Agarwal and Fung. Multiple fault testing of large circuits by single fault test sets. *IEEE Transactions on Computers*, C-30(11):855–865, 1981. doi: 10.1109/TC.1981.1675716.

[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi: 10.1109/TDSC.2004.2.

[10] Marcello Barbirotta, Antonio Mastrandrea, Francesco Menichelli, Francesco Vigli, Luigi Blasi, Abdallah Cheikh, Stefano Sordillo, Fabio Di Gennaro, and Mauro Olivieri. Fault resilience analysis of a risc-v microprocessor design through a dedicated uvm environment. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020. doi: 10.1109/DFT50435.2020.9250871.

[11] A. Benso, A. Bosio, S. Di Carlo, and R. Mariani. A Functional Verification based Fault Injection Environment. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 114–122, 2007. doi: 10.1109/DFT.2007.31.

[12] J. Carreira, H. Madeira, and J.G. Silva. Xception: a technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998. doi: 10.1109/32.666826.

[13] Yung-Chang Chang, Li-Ren Huang, Hsing-Chuang Liu, Chih-Jen Yang, and Ching-Te Chiu. Assessing automotive functional safety microprocessor with iso 26262 hardware requirements. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, pages 1–4, 2014. doi: 10.1109/VLSI-DAT.2014.6834876.

[14] A. C. L. Chiang, I. S. Reed, and A. V. Banes. Path sensitization, partial boolean difference, and automated fault diagnosis. *IEEE Transactions on Computers*, C-21 (2):189–195, 1972. doi: 10.1109/TC.1972.5008925.

[15] Avidan Efody. Getting ISO 26262 Faults Straight. URL `https://verificationacademy.com/topics/functional-safety/articles/Getting-ISO-26262-faults-straight`. Accessed: 2023-07-14.

[16] Sameh El-Ashry, Mostafa Khamis, Hala Ibrahim, Ahmed Shalaby, Mohamed Abdelsalam, and M. Watheq El-Kharashi. On error injection for noc platforms: A uvm-based generic verification environment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(5):1137–1150, 2020. doi: 10.1109/TCAD.2019.2908921.

[17] International Organization for Standardization (ISO). Road vehicles – Functional safety, ISO 26262, 2011.

[18] International Organization for Standardization (ISO). Road vehicles – Functional safety - Part 1: Vocabulary, ISO 26262, 2011.

[19] International Organization for Standardization (ISO). Road vehicles – Functional safety - Part 5: Product development at the hardware level, ISO 26262, 2011.

[20] Gajski and Kuhn. Guest editors' introduction: New VLSI tools. *Computer*, 16(12): 11–14, 1983. doi: 10.1109/MC.1983.1654264.

[21] Robert Bosch GmbH. Bosch MEMS sensors, 2022. URL `https://www.bosch-mobility.com/en/solutions/electronic-components/mems-sensors/`. Accessed: 2023-07-12.

[22] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–347, 1989. doi: 10.1109/FTCS.1989.105590.

[23] J.L.A. Hughes. Multiple fault detection using single fault test sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(1):100–108, 1988. doi: 10.1109/43.3137.

[24] J. Karlsson, U. Gunneflo, P. Liden, and J. Torin. Two fault injection techniques for test of fault handling mechanisms. In *1991, Proceedings. International Test Conference*, pages 140–, 1991. doi: 10.1109/TEST.1991.519504.

[25] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–23, 1994. doi: 10.1109/40.259894.

[26] Philipp Kilian, Armin Köhler, Patrick Van Bergen, Markus Wörz, Martin Schneider, Thorsten Groh, Tihomir Tomanic, and Martin Dazer. Best practices for advanced modeling of safety mechanisms in an fta. *IEEE Access*, 11:60109–60129, 2023. doi: 10.1109/ACCESS.2023.3284751.

[27] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2014. doi: 10.1109/DTIS.2014.6850649.

[28] Douglas Lohmann, Fabrizio Maziero, Elco João dos Santos, and Djones Lettnin. Extending universal verification methodology with fault injection capabilities. In *2018 IEEE 9th Latin American Symposium on Circuits Systems (LASCAS)*, pages 1–4, 2018. doi: 10.1109/LASCAS.2018.8399945.

[29] G. Miremadi and J. Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44(3): 441–454, 1995. doi: 10.1109/24.406580.

[30] Daniel Mueller-Gritschneder, Petra R. Maier, Marc Greim, and Ulf Schlichtmann. System c-based multi-level error injection for the evaluation of fault-tolerant systems. In *2014 International Symposium on Integrated Circuits (ISIC)*, pages 460–463, 2014. doi: 10.1109/ISICIR.2014.7029567.

[31] Alessandra Nardi, Samir Camdzic, Antonino Armato, and Francesco Lertora. Design-For-Safety for automotive IC design: Challenges and opportunities. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, 2019.

[32] Animesh Sarkar. ASIL (automotive safety integrity levels) ratings simplified, 2022. URL `https://www.linkedin.com/pulse/asil-automotive-safety-integrity-levels-ratings-animesh-sarkar/`. Accessed: 2023-07-13.

[33] Horst Schirmeier, Martin Hoffmann, Christian Dietrich, Michael Lenz, Daniel Lohmann, and Olaf Spinczyk. Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015*

11th European Dependable Computing Conference (EDCC), pages 245–255, 2015. doi: 10.1109/EDCC.2015.28.

[34] V. Sieh, O. Tschache, and F. Balbach. Verify: evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 32–36, 1997. doi: 10.1109/FTCS.1997.614074.

[35] Daniel Skarin, Raul Barbosa, and Johan Karlsson. Goofi-2: A tool for experimental dependability assessment. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 557–562, 2010. doi: 10.1109/DSN.2010.5544265.

[36] Loughborough University. An introduction to MEMS, 2002. URL `https://www.lboro.ac.uk/microsites/mechman/research/ipm-ktn/pdf/Technology_review/an-introduction-to-mems.pdf`. Accessed: 2023-07-12.

[37] P Vinopoornima and G Dhanabalan. Identification of stuck-at-faults of full adder using fpga as a testing device. In *2019 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, pages 1–4, 2019. doi: 10.1109/INCOS45849.2019.8951350.

[38] Jeff Watson. MEMS gyroscope provides precision inertial sensing in harsh, high temperature environments, 2016. URL `https://www.analog.com/en/technical-articles/mems-gyroscope-provides-precision-inertial-sensing.html`. Accessed: 2023-07-13.