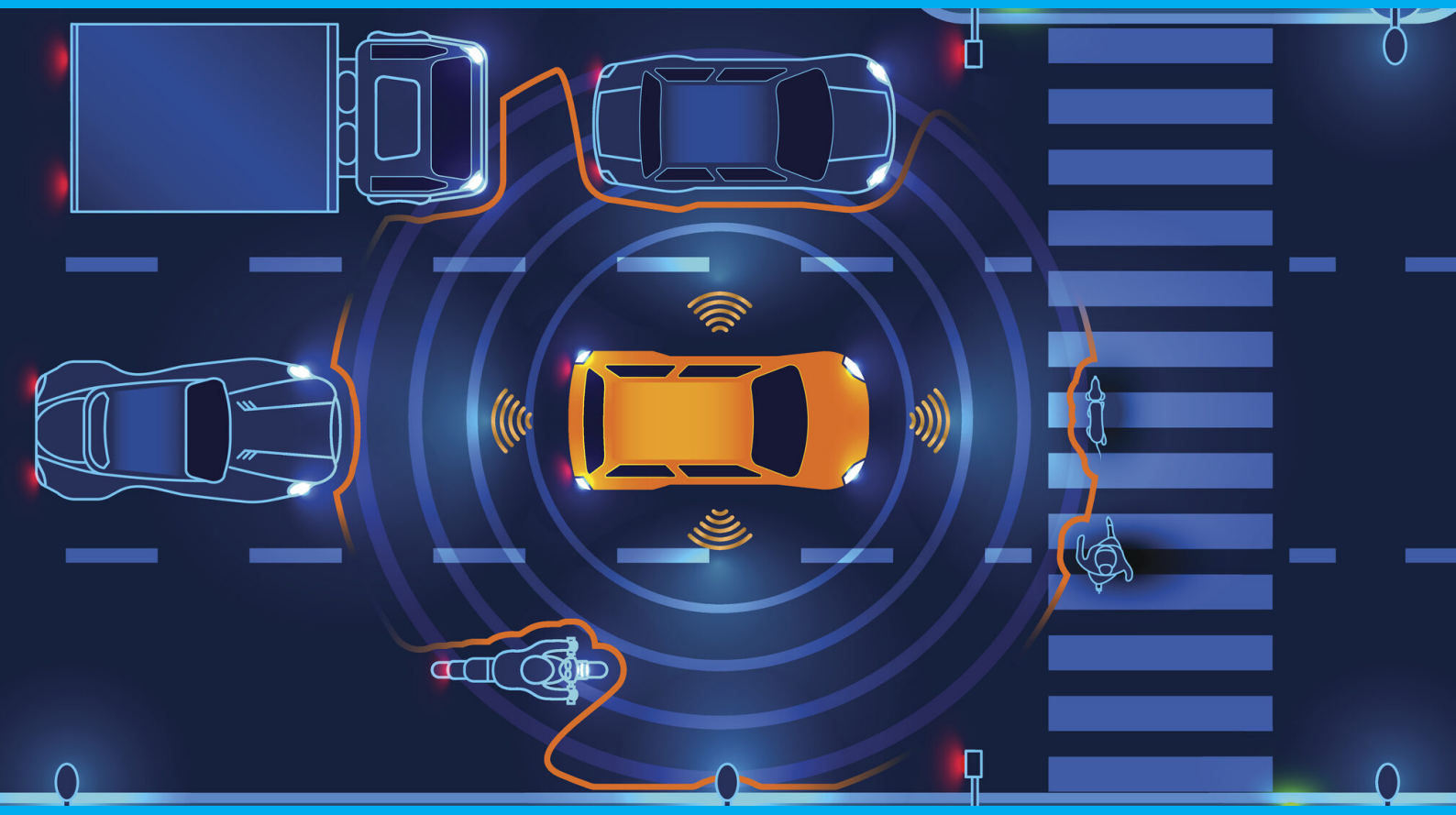
Optimising Functional Safety EDA for Accurate Fault Coverage Estimation

Abhiroop Bhowmik



TU Delft

Optimising Functional Safety EDA for Accurate Fault Coverage Estimation

by

Abhiroop Bhowmik

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday May 14, 2024 at 9:30 AM

Student number:	5684986	
Project duration:	July 17, 2023 - May 14, 2024	
Thesis committee:	Dr. Ir. M. Taouil,	TU Delft, supervisor
	Dr. Ir. M. C. R. Fieback,	TU Delft, supervisor
	Dr. Ir. C. Gao,	TU Delft
	Ir. S. Babukutty,	NXP Semiconductors

In science, the credit goes to the man who convinces the world,
not to the man to whom the idea first occurs.

Francis Darwin

Abstract

The automotive industry is experiencing a significant shift towards advanced electronic and software integration, driven by the increasing demand for self-driving and autonomous vehicles. With electronics now making up a major portion of vehicle costs, ensuring their reliable operation is critical. However, as the complexity of automotive systems increases, so do the risks associated with malfunctions, requiring a critical need for robust safety measures. Functional Safety (FuSa), as defined by ISO 26262, provides a framework for addressing these concerns at different stages of the safety lifecycle. The primary aim of FuSa is to develop Safety Mechanisms (SM) to detect faults and recover from them. The efficiency of these SMs is indicated by Diagnostic Coverage (DC), which represents the percentage of detected faults. In this context, there are several challenges in verifying the functional safety of automotive chips, especially with RTL designs. For example, identification of safe faults is one of the initial steps in FuSa verification. Discrepancies are observed in their classification when utilising different techniques such as Automatic Test Pattern Generation (ATPG), formal methods and fault injection simulation. This raises questions about the accuracy of overall results obtained from these tools as well. Varying outcomes from fault simulation EDA tools in classifying faults may result in different Automotive Safety Integrity Levels (ASIL) assigned to the component being assessed. This discrepancy would misrepresent the component's ability to reduce associated risks, highlighting the importance of conducting a detailed analysis and comparison of the tools.

The thesis provides a comprehensive evaluation of EDA tools utilized for Functional Safety Verification, focusing on RTL designs. Scripts are developed to automate fault simulation flows of two prominent FuSa EDA tools, XFS by Cadence and VC Z01X by Synopsys, and derive automatic comparisons. By comparing these tools, their strengths and limitations are analyzed. XFS exhibits limitations in fault propagation on input and output ports, resulting in the omission of certain fault scenarios. VC Z01X showcases faster fault simulation capabilities along with an extensive feature set for fault simulation, but lacks support for transient fault injection on a section of the fault subspace. By applying the automated tool flows on a FIFO design enabled with ECC, the DC obtained from XFS and VC Z01X are 68.96% and 80.47% respectively, showcasing a major difference. These findings highlight the importance of a holistic verification methodology that accurately estimates diagnostic coverage.

A novel verification methodology is proposed, which combines the strengths of XFS and VC Z01X to optimize the efficiency and accuracy of fault simulation. Leveraging VC Z01X's concurrent engine for parallel fault injection and XFS's capabilities to cover the unexplored fault space, this integrated approach provides comprehensive fault coverage. The flow also provides users the capability to update fault classification results based on manual analysis or designer inputs, thereby changing the DC as well. The verification methodology is applied to the AutoSoC benchmark suite, an automotive System-on-Chip with configurable SMs. Based on the results, additional SMs are implemented in the AutoSoC design - duplication of pipeline stages with temporal redundancy and ECC on internal memories. This leads to an estimated area increase of 1.4x as compared to the baseline design, but also results in the qualification of an ASIL C level component with a DC of 97.79%. The baseline verification flow included in the benchmark suite provides a DC of 98.36%, which is an over estimation of the actual coverage. The proposed methodology provides a more accurate coverage by taking into account the maximum possible fault space and considering transient faults as well. While there remains room for further improvement in verification methodologies, this framework effectively addresses the fault space required for FuSa verification and provides an accurate estimation of Diagnostic Coverage.

Acknowledgements

I would like to thank the following people without whom I would not have been able to complete my thesis:

- First and foremost, I would like to thank my parents, whose support has been instrumental in my academic and personal journey. To my father especially, who has taken care of the smallest things since my childhood and who survived a big scare during the course of my thesis. His road to recovery has been a constant reminder of all the effort that he has put for me throughout his life for which I would like to thank him whole heartedly.
- Second, I would like to thank my supervisors at TU Delft, Dr. Ir. Mottaqiallah Taouil and Dr. Ir. M.C.R. Fieback, who provided me immense support during the entire course of the thesis by providing me valuable feedback on the work, giving me direction, and encouraging me to look at things critically during the course of the thesis. The weekly discussions with them allowed me to reflect on my work and constantly strive to improve upon it.
- I would also like to thank my supervisors at NXP Semiconductors, Ir. Subin Babukutty, Ir. John Dielissen and Ir. Jan Vink for their continued mentorship and guidance throughout the thesis, and for providing me support through challenging times. Their insightful feedback on my work, along with my report, has not only enhanced my understanding of Functional Safety Verification but also improved my ability to critically analyze concepts, contributing significantly to the completion of my thesis.
- Last, but not least, I would like to thank my partner, Manali Shah, who has been my constant support throughout this entire period and who I could rely on for the smallest of things. To my friends with whom I have interacted over the past two years and spent my time with, their company and support have also kept me going through this thesis, and a big shoutout to them for providing me the laughs and giggles!

Abhiroop Bhowmik,
Delft, May 2024

List of Acronyms

EDA Electronic Design Automation

FuSa Functional Safety

RTL Register Transfer Level

GLN Gate Level Netlist

HARA Hazard Analysis and Risk Assessment

ASIL Automotive Safety Integrity Level

FMEDA Failure Mode Effect and Diagnostic Analysis

FTA Fault Tree Analysis

DFA Dependent Failure Analysis

DC Diagnostic Coverage

FTTI Fault Tolerant Time Interval

SPFM Single Point Fault Metric

LFM Latent Fault Metric

PMHF Probabilistic Metric for Random Hardware Failures

SA Stuck-At

SEU Single Event Upset

SET Single Event Transient

FM Failure Mode

SM Safety Mechanism

FIT Failure in Time

COI Cone of Influence

TCL Tool Confidence Level

ATPG Automatic Test Pattern Generation

DCLS Dual Core Lock Step

EDC Error Detection Code

ECC Error Correction Code

SECDED Single Error Correction Double Error Detection

STL Software Test Library

TMR Triple Modular Redundancy

BIST Built-in Self Test

CRC Cyclic Redundancy Check

XFS Xcelium Fault Simulator (Cadence)

VC Verification Continuum (Synopsys)

SFF Standard Fault Format

FCC Fault Campaign Compiler

FCM Fault Campaign Manager

UDFS User Defined Fault Status

FIFO First In First Out

SDPRAM Simple Dual Port Random Access Memory

FID Fault ID

UVM Universal Verification Methodology

JG FSV JasperGold Functional Safety Verification

AutoSoC Automotive System-on-Chip

CAN Controller Area Network

LIN Local Interconnect Network

AES Advanced Encryption Standard

DES Data Encryption Standard

JTAG Joint Test Action Group

UART Universal Asynchronous Receiver/ Transmitter

ALU Arithmetic Logic Unit

LSU Load Store Unit

WB WishBone

RF Register File

MMU Memory Management Unit

RTEMS Real-Time Executive for Multiprocessor Systems

HDL Hardware Description Language

OD Observed Diagnosed - **O**bserved at Functional strobe, **D**iagnosed at Checker strobe

ON Observed Not Diagnosed - **O**bserved at Functional strobe, **N**ot diagnosed at Checker strobe

ND Not Observed Diagnosed - **N**ot observed at Functional strobe, **D**iagnosed at Checker strobe

NN Not Observed Not Diagnosed - **N**ot observed at Functional strobe, **N**ot diagnosed at Checker strobe

IX Impossible x-state - transient faults injected on signals at x-state

NC Not Controllable - faults on locations which do not toggle during simulation

UU Untestable Unused - Unused signals considered Safe

Contents

List of Figures

List of Tables

1

Introduction

1.1. Motivation

The automotive industry has undergone significant transformation, marked by a notable surge in the integration of electronic components and software, driven in part by the escalating demand for self-driving and autonomous vehicles. In the 1970s, electronics constituted merely 5% of a vehicle's total composition [1], and by 2020, it accounted for 35% of overall vehicle costs [2]. Projections suggest a further elevation to 50% by the end of 2030. Illustrated in Figure 1.1, the ascending trajectory of the automotive electronics market is evident, with its global valuation reaching USD 289 billion in 2022, and anticipated to exhibit a compound annual growth rate (CAGR) of 7.8% from 2023 to 2032 [3]. This surge is fueled by the extensive integration of advanced safety systems, including automatic airbags, anti-lock braking mechanisms, parking assist systems, emergency braking, and lane warning systems, all aimed at mitigating road accidents [4]. Moreover, the incorporation of features such as alcohol ignition interlocks, emergency call systems, and accident data recording systems is rapidly gaining prominence for vehicular protection, thus poised to propel industry expansion throughout the ensuing decade [4].

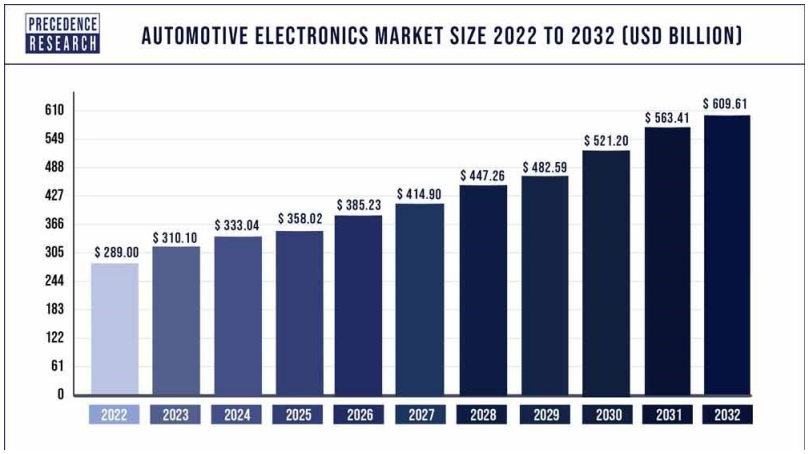


Figure 1.1: Predicted growth in Global Automotive Electronics Market [3]

The expansion of electronic components in vehicles introduces an increased vulnerability to failures stemming from environmental conditions, electromagnetic interference, wear and tear, and latent bugs in the hardware or software of these components. Such failures can bring about hazardous events over a vehicle's lifespan, posing life-threatening risks to its users. Figure 1.2 emphasizes the magnitude of global vehicle recalls, reaching as high as approximately 15 million, primarily attributable to defects in Integrated Electronic Components (IECs) and software issues [5]. This begs the question as to what kind of measures and practices could be brought into place and enforced in order to stop this number from going even further upwards, and ensure utmost safety in vehicular functionality.

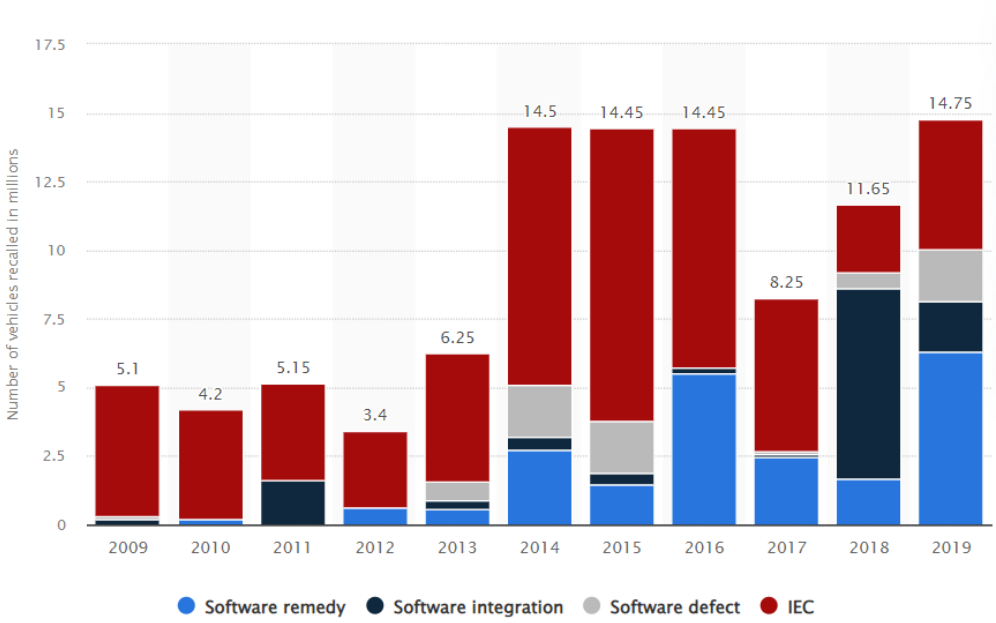


Figure 1.2: Global vehicle recalls till 2019 [5]

This is where the concept of Functional Safety emerges to ensure that an automotive system does not lead to hazardous situations even when unexpected errors occur during it's operation. Functional safety, as defined by ISO26262 [6], is "the absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E [1] systems,". This serves as a foundational approach to securing the safety of automotive electronics, further explained in Chapter 2. Basic questions such as, "Can I trust my car's automatic electronic steering wheel to function properly if a component unexpectedly fails while driving?" or "What are the implications if the anti-lock braking system malfunctions on a high-speed highway?" need to be answered. In essence, functional safety provides the necessary framework to address such concerns and guarantee the safe operation of vehicles amidst potential system failures.

The ISO 26262 standard, an adaptation of the IEC 61508 series [7], addresses the sector-specific needs of electrical and electronic systems in road vehicles with respect to functional safety. It guides the diverse activities undertaken during the development and lifecycle of safety-related systems comprising electrical, electronic, and software components. Adherence to ISO 26262 is imperative for ensuring the functional safety of automotive chips during their hardware development, validating their suitability for real-world deployment.

The Electronic Design Automation (EDA) industry assumes a pivotal role in facilitating the development of ISO 26262-compliant automotive chips. With an array of tools tailored for various stages of chip development, EDA tools, particularly those focused on Functional Safety Verification, play a critical role. Functional Safety verification extends beyond traditional functional verification, requiring compliance with various safety requirements throughout different stages of the development lifecycle and is thus one of the most critical issues in automobile development. This thesis delves into an exploration of different EDA tools, aiming to provide a solution that contributes to the thorough functional safety verification of chip designs tailored to meet automotive requirements.

1.2. Problem Statement

On a high level, functional safety involves introducing faults in the design of an electronic component, and understanding their effect on the functional outputs of the system. Safety mechanisms must be implemented in the design to detect faults and initiate recovery mechanisms. The effectiveness of these Safety Mechanisms is determined by Diagnostic Coverage, indicating the proportion of detected faults. Diagnostic Coverage establishes the extent of risk reduction achievable by the design. In this thesis,

[1] Electrical and Electronic

we will be looking at digital designs with Register Transfer Level (RTL) or Gate Level Netlist (GLN) descriptions, and verify their functional safety with respect to ISO 26262. But, what are the problems in current functional safety solutions for the same?

While leading vendors such as Cadence, Synopsys, and Siemens (Mentor Graphics) offer dedicated EDA tools for Functional Safety Verification [8], an evident dearth of empirical research persists on the rationale behind favoring one tool over another. Moreover, the reliability and accuracy of tool-generated results pose inherent challenges, prompting scrutiny of the tools themselves. For example, identification of safe faults is one of the preliminary steps in functional safety verification and [9], [10] discuss utilising different technologies such as ATPG, formal methodologies and fault injection simulation to identify the same. Discrepancies are seen with the usage of different tools and technologies which need to be carried out manually. Continuing on the same note, if we use two tools from different vendors to carry out functional safety verification on a single design, will they provide the same overall results? If not, which of these results do we eventually trust?

Ensuring functional safety involves injecting various types of faults and accurately classifying them to calculate the desired fault coverage. Therefore, it is critical to consider the tools' capability to cover the fault space effectively and give correct classification results. Further, it is crucial to start the functional safety verification process as soon as possible in the safety lifecycle, preferably at the RTL development stage. The duration required for finalizing the RTL and creating the GLN can be lengthy. Discovering bugs or functional safety issues at a later stage may consequently extend the time to market [11]. Thus, the main question focuses on finding better ways to make sure the Functional Safety Verification process is strong and reliable through improved verification methods. In light of these considerations, we propose the following research question:

"Are there discrepancies in the results of FuSa EDA tools at the RTL stage? If so, how can they be resolved to provide accurate diagnostic coverage estimations?"

In order to answer our primary research question, we need to look at the following sub-topics in order to understand it better.

1. To address the question of discrepancies, a thorough examination of current Functional Safety EDA tools is necessary. This involves a comparative analysis of features and performance metrics to understand the strengths and limitations of each tool.

2. A verification methodology must be developed by considering the comparison of the tools, aiming to address their limitations, if any. The determination of whether one tool significantly outperforms the other, or if a combinations of various approaches is preferable, depends on the findings from the comparison process.

3. Following the verification process, a detailed review presenting relevant metrics essential for ISO 26262 qualification needs to be provided. This includes a detailed analysis of the design, diagnosing areas that require improvement in safety mechanisms and offering insights into achieving increased Functional Safety. It should also be able to provide suggestions for improving Safety Mechanisms and refining the overall chip design.

1.3. Thesis Contributions

The thesis aims to develop a complete and robust verification flow which can be used to test the Functional Safety of any given chip, with a concurrent emphasis on proposing enhancements. In this regard, the major contributions of the thesis are as follows:

1. An in-depth comparative analysis of EDA tools from Cadence and Synopsys for Functional Safety Simulation is carried out in order to figure out what works better and can be integrated in the verification flow of any required design at the RTL stage. Key comparison metrics are defined to evaluate the performance of each tool. The tool flows are tested with small reference designs to analyse the working and results of the tools.

2. Scripts are developed in order to automate the verification flows of both the tools. Additionally, post-processing scripts are developed to extract differences in the results along with other necessary information in order to facilitate easy debug and reduce manual work. This is further tested on a reference design featuring Safety Mechanisms, and is thus relevant for deriving the defined comparison metrics.

3. Following the evaluation of tools, a unified verification methodology is proposed using a combination of the two tools with additional features and utilities incorporated using scripts. The primary goal of this framework is to comprehensively address the fault space, providing precise results and metrics. It is adaptable and can be utilized in diverse verification environments with minimal changes based on the specific design.

4. The proposed verification flow is applied to the AutoSoC open-source automotive design, serving as a case study. Results are analyzed to identify areas for improvement, leading to enhancements within the design. Contributions are made to this open source design, by implementing ECC on all internal memories of the design, along with the duplication of pipeline stages with temporal redundancy. The inclusion of these hardware Safety Mechanisms improves the Functional Safety metrics of the AutoSoC without imposing significant area overhead, as is often the case with other Safety Mechanisms, like Dual Core Lock Step processors, providing similar metrics.

5. The aforementioned contributions are consolidated in a scientific paper discussing the proposed unified verification methodology and its results (to be submitted to VLSI SoC, 2024).

1.4. Report Outline

This thesis is organized into seven chapters. Chapter 2 provides an introduction to the key concepts of Functional Safety Verification, briefing various parts of the ISO 26262 standard. It outlines the safety lifecycle to be adhered to and highlights important metrics and results to be considered upon completion of the verification flow. In Chapter 3, the literature survey and an exploration of the current state-of-the-art in Functional Safety Verification are presented. This section offers insights into diverse methodologies and techniques employed in the verification process, with a motivation to enhance existing practices using established Safety Mechanisms. Chapter 4 offers an overview of existing EDA tool flows for fault injection simulation. It also provides a comparative analysis of tools applied to a design (FIFO) with Safety Mechanisms (ECC and module duplication), presenting results in relation to predefined metrics. Building upon the findings outlined in this section, Chapter 5 introduces a verification methodology aimed at maximizing the utility of the discussed tools to develop a reliable and robust verification solution. Subsequently, Chapter 6 delves into the outcomes of the proposed verification flow applied to the FIFO design and extends the analysis to include AutoSoC, a more applicable open-source automotive SoC design. This chapter also explores the integration of additional Safety Mechanisms to improve the design following the verification flow and assess possible enhancements. Finally, Chapter 7 offers concluding remarks on the thesis study and explores avenues for future research and enhancements in Functional Safety Verification.

2

Background

2.1. Functional Safety (FuSa) Verification with regards to ISO 26262

The primary aim of Functional Safety is to establish a reliable system capable of operating as intended, even in the face of accidental or unexpected circumstances. It also aims to mitigate the level of risk or injury resulting from such occurrences. As stated previously, adherence to ISO 26262 is imperative to ensure functional safety throughout the product lifecycle, covering aspects such as specification, design, verification and production. To achieve this goal, it is crucial to implement safety systems or mechanisms within the product, typically accomplished through the utilization of two techniques:

- **Error detection systems** belong to the category of safety mechanisms, designed with checkers to monitor the system and activate responses as needed, triggering relevant recovery features. Such systems primarily offer partial recovery capabilities.
- **Error detection and correction systems** can be developed using redundant architectures, which would provide multiple copies of the same logic/system that would limit the risk of any error that upsets the system. While these systems consume more area compared to error detection systems, they offer increased recovery capabilities.

The root cause of having to develop functionally safe systems arises from the fact that failures could happen in any system, because of various reasons such as radiation sources, electromagnetic fields, and normal wear and tear of systems. ISO 26262 defines two classes of failures:

- **Systematic failures** are failures which are essentially created during design, also known as bugs. The causes of these failures can be tracked in a deterministic way, and would require a change in the design process, verification or any of the stages of typical product development in order to eliminate the same. It is necessary to have a good design flow to avoid such failures. Such a design flow would entail the creation of specification based on requirements, design, verification, prototyping, validation and evaluation. It is paramount to have standardized reviews at each stage of this design process to eliminate human errors and bugs to avoid systematic failures.
- **Random failures** are failures which generally happen after manufacturing, whose root causes are probabilistic and can occur anytime during the lifetime of a hardware device. As mentioned before, the usual causes of such failures are electromagnetic radiation, aging, heat, temperature, technology processes, among others. Random failures cannot be completely prevented and hence safety mechanisms need to be deployed in order to mitigate the effects of such failures.

The ISO 26262 standard defines all the necessary guidelines and measures which need to be taken to assess the safety mechanisms and procedures developed to mitigate the effects of both categories of failures. As shown in Figure 2.1, the latest ISO 26262 standard, which was released in 2018, defined a total of 12 parts in the entire safety lifecycle to standardize the Functional Safety process.

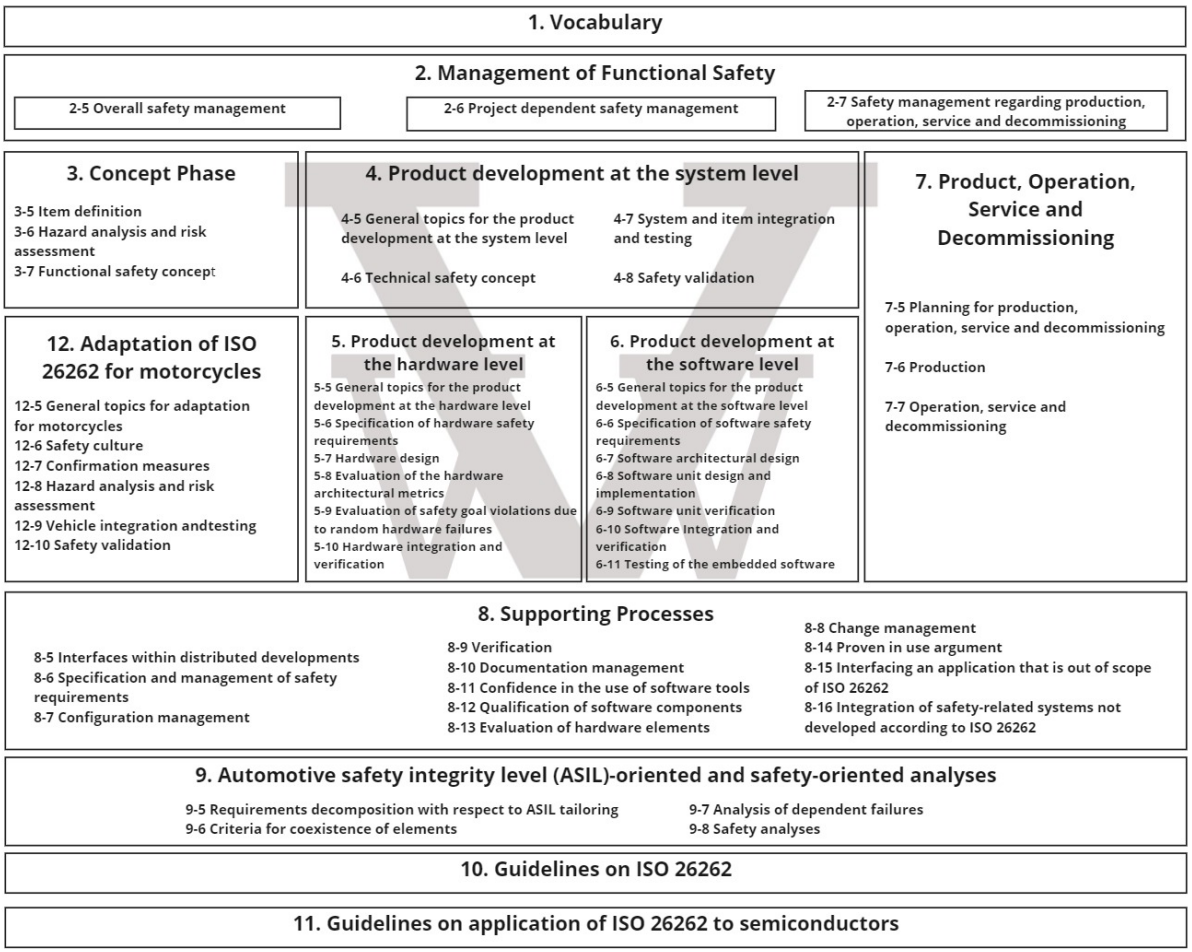


Figure 2.1: Overview of ISO 26262 safety lifecycle [6]

Every part in ISO 26262 has four general topics - Scope, normative references, terms and definitions, and requirements for compliance. Hence, as shown in Figure 2.1, the numbering of the subsections in each part starts from 5 onward. A quick overview of the individual parts is listed below to get an overall picture of ISO 26262:

- Part 1 highlights common vocabulary and definitions to establish clarity and coherence for the remainder of the standard.
- Part 2 outlines an extensive approach to managing functional safety throughout the project lifecycle, covering overall safety management and project-specific directives.
- Part 3 focuses on the conceptual phase, focusing on topics such as Hazard Analysis and Risk Assessment (HARA), defining Functional Safety Requirements, and establishing Safety Goals. Automotive Safety Integrity Levels (ASIL) are defined for components at this stage.
- Part 4 addresses system-level product development, including technical safety specifications and top-level system architectural design.
- Part 5 dives into hardware development, encompassing hardware design and the evaluation of safety goal violations, along with determining relevant metrics for classification of automotive components.
- Part 6 specifies software development guidelines, covering software architectural design, unit design, and verification, and the additional steps to be followed in comparison to the typical product development of software.
- Part 7 guides production, operation, service, and decommissioning processes for safety-related elements in road vehicles, and deals with safety processes after the completion of system development.

- Part 8 governs supporting processes, ensuring verification, tool qualification, and proven in-use arguments are correctly executed.
- Part 9 is ASIL-oriented, and focuses on ASIL decomposition of different components, coexistence criteria for such elements, and dependent failure analysis.
- Part 10 offers guidelines to enhance understanding and application of ISO 26262.
- Part 11 provides detailed support for semiconductor manufacturers and integrators, with additional guidelines on topics from the previous parts.
- Lastly, Part 12 outlines the adaptation of ISO 26262 for motorcycles, covering safety culture, confirmation measures, hazard analysis, vehicle integration, and safety validation.

In the next section, we discuss each of the parts briefly, with detailed discussion on relevant sections for deeper understanding of the background knowledge required for this thesis.

2.2. Parts of ISO 26262

2.2.1. Part 1 - Vocabulary

Part 1 - Vocabulary encompasses the formal definitions, terms and abbreviations, which would be applied in all parts of the standard. The functional safety lifecycle starts at an "Item", which is defined as a "system or combination of systems to which ISO 26262 is applied, that implements a function or part of a function at the vehicle level". It is further broken down into elements, which could be a system, component or any hardware/software unit. A system should relate a sensor, controller and an actuator with one another.

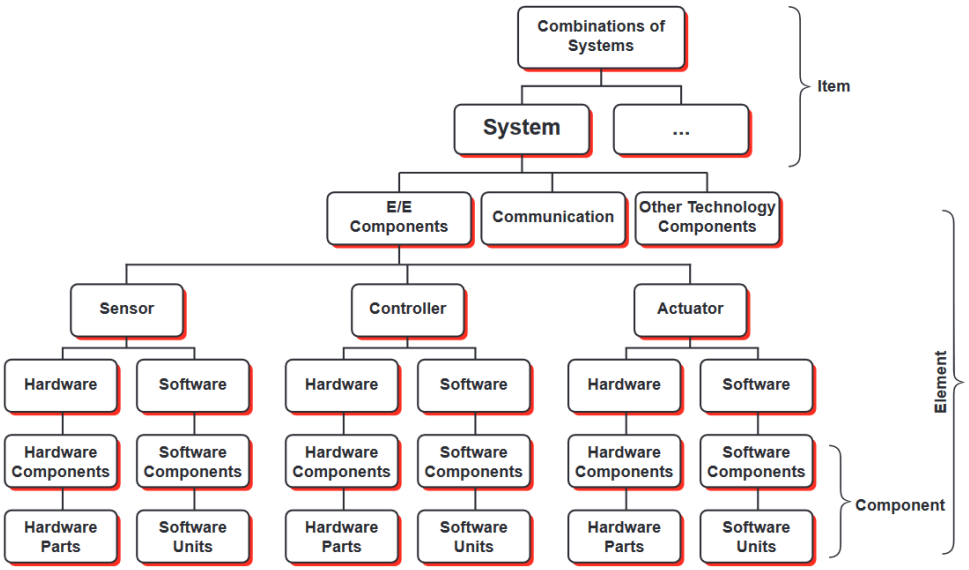


Figure 2.2: Terminologies in ISO 26262 [12]

Illustrating the application of the introduced terms, we can take the example of an Anti-Lock Braking System (ABS), whose function is to prevent locking of the wheel and provide more control to the driver. In the context of the standard, ABS will be an Item. Different elements of the ABS item could be E/E system (ABS Control Unit, Sensing Unit, Pump Motor Control Unit, Valve Control Unit), Mechanical System (Actual Brake pedals, Brake Pads, Rotor, Brake Hose) and Hydraulic System (Master Cylinder, Accumulator, Reservoir, Pump Motor, Piston). Figure 2.2 shows a perspective of the hierarchy of these terminologies in ISO 26262. It gives an overview of the overall system to which the safety lifecycle is applied. In the context of the thesis, we would be looking into individual E/E components to exercise functional safety verification.

2.2.2. Part 2 - Management of Functional Safety

Part 2 - Management of Functional Safety addresses the safety lifecycle and compliance with the standard in terms of three clauses. *Overall Safety Management* is related to the organization which develops

the product and includes procedures such as developing a company-specific lifecycle, definition of tools to be used, different kinds of safety analyses to be performed, among other things [13]. The second clause is regarding *Safety Management at the Project Level*, which outlines the actions required throughout product development, particularly focusing on impact analysis. The outcomes of this analysis are utilized to customize the lifecycle and formulate the necessary safety activities. The third and final clause is about *Safety Management post Development*, and talks about processes which need to be in place regarding the safety of production and in-field monitoring. To summarise, the second part of ISO 26262 discusses the comprehensive management of the safety lifecycle, specifically tailored to the development organization. It further explores the necessary measures to be implemented during and after the development phase, ensuring the ongoing maintenance of the product's functional safety.

2.2.3. Part 3 - Concept Phase

Part 3 - Concept Phase is the initial phase in product development wherein different concepts start to become more concrete. Essential product functionalities must be outlined, accompanied by a risk analysis to allocate specific levels of importance to the system. Finally, a detailed functional safety concept must be articulated to understand all system requirements. This section is further subdivided into three clauses -

1. **Item Definition**: As previously noted, an Item serves as a pivotal starting point in product development. It represents the primary subject of development — the product and it is important to clearly define and delimit the scope of the intended functionalities and constraints.

2. **Hazard Analysis and Risk Assessment**: ISO 26262 stresses the significance of understanding risks associated with the product and identifying potential hazardous situations it may lead to. The standard proposes a Hazard Analysis and Risk Assessment (HARA) strategy to identify different hazards, assess the associated risks and then accordingly formulate safety goals. HARA begins with a description of operational situations and modes of the product, and the hazards which could arise in the event of a fault in the specified Item. A hazard is defined as "a potential source of harm caused by malfunctioning behavior of the item". It is important to understand the operational situations to truly assess the risk of the item. For instance, taking an ABS system into consideration, the impact of a hazardous event would vary if it occurred within a garage as opposed to transpiring on the highway, where the consequences could potentially be life-threatening. So, an in-depth risk analysis needs to be performed and then the corresponding "Automotive Safety Integrity Level" or "ASIL" is assigned. ASIL is determined in terms of four levels, each level designated with a letter, A being the lowest and D being the highest.

   To conduct the risk analysis, it is essential to evaluate three metrics associated with the pertinent hazardous events - Severity, Exposure, and Controllability.

   - **Severity** is quantified in terms of the harm caused by the event, considering factors such as potential injuries and their extent. This metric spans from level S1 to S3, where S1 denotes light injuries, and S3 indicates life-threatening or fatal injuries. Sometimes, an S0 level is also included, signifying no injuries.
   - **Exposure** is determined by the likelihood or probability of the hazard, relying on statistical information obtained from diverse sources related to similar vehicles and systems, corresponding technologies, and the rate of traffic accidents. As shown in Table 2.1, it is expressed in 4 levels ranging from E1 to E4, with E4 being the highest probability of occurrence.
   - **Controllability** is evaluated based on the driver's ability to manage an unwanted situation and prevent a hazardous event. This metric is also categorized into three levels, ranging from C1 to C3. A C1 level signifies that the situation is easy to control, whereas a C3 level indicates that it is very challenging to control. Additionally, there is sometimes a C0 level associated with this metric, indicating that the situation is generally controllable.

   The combination of these metrics gives rise to the corresponding ASIL level of an item, ranging from A to D, as described by the matrix shown in Figure 2.3. There is also an additional level called QM (Quality Management), in which the development of the item is sufficient according to the established quality management at the organization. As depicted in the figure, as the levels

Table 2.1: Exposure probability table [14]

Class	E1	E2	E3	E4
Description	Very low probability	Low probability	Medium probblity	High probability
Definition of frequency	Situations that occur less often than once a year for the great majority of drivers	Situations that occur a few times a year for the great majority of drivers	Situations that occur once a month or more often for an average driver	All situations that occur during almost every drive on average

Table 2.2: Classification of controllability [14]

Class	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable
Definition	Controllable in general	99% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	90% or more of all drivers or other traffic participants are usually able to avoid a specific harm.	Less than 90% of all drivers or other traffic participants are usually able, or barely able, to avoid a specific harm.

of the metrics increase, the likelihood of a life-threatening hazard also rises, and the ability to control such a scenario becomes more difficult. Consequently, a higher ASIL level is then assigned.

Figure 2.3: Determination of ASIL level in ISO 26262 using HARA[15]

Continuing with the example of an ABS, the primary function of this system is to prevent wheel

locking during braking and thereby maintaining tractive contact with the surface of the road. Let us consider a hazardous situation wherein there is a loss of control on a split-μ surface (Surfaces with asymmetrical friction coefficient) [16]. The severity of this hazard would then be assigned a level of S3, as there could be fatal injuries in case of an accident triggered by this hazardous situation. The probability of this scenario occurring could also be high on typical highways, and thus, the exposure level is E4. Finally, this kind of a situation can be difficult to control because of the automatic engagement of ABS in an unwanted scenario, and hence, the controllability is C3. Combining these three metrics, we get an ASIL level of D for this particular feature. Once the ASIL level is determined, appropriate **safety goals** need to be formulated, which in our case could be as follows:

- Additional sensors need to be in place to determine the difference in friction coefficient to provide better control.
- Improved steering mechanisms need to be developed to allow added control from the driver.

These safety goals must be implemented and developed later in the course of the lifecycle. Subsequently, the relevant metrics, as defined by the hardware development section, need to be met, as will be elaborated upon later. To provide a perspective on the ASIL levels of different systems in an automotive context, an example is shown in Figure 2.4.

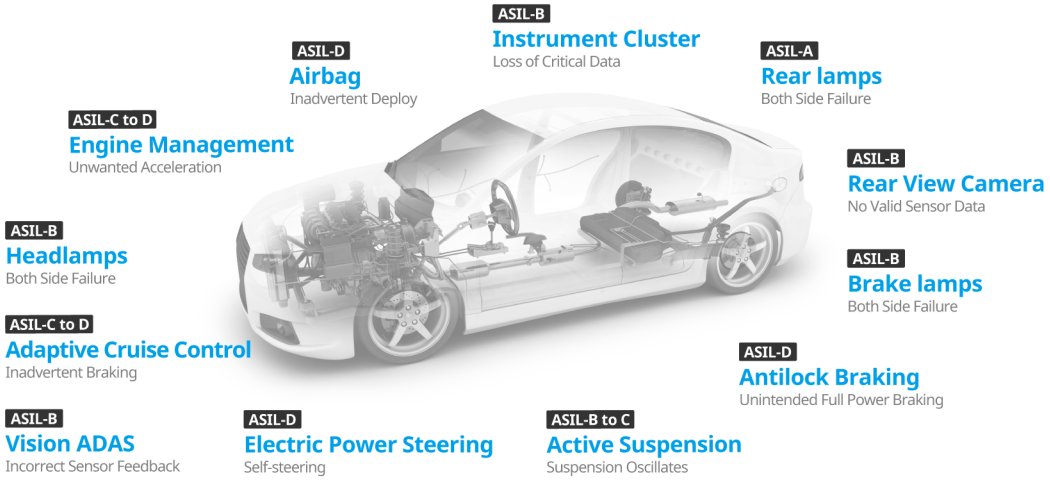


Figure 2.4: ASIL levels of different components in an automotive [17]

3. **Functional Safety Concept**: This clause discusses the derivation of functional safety requirements from the safety goals, thereby providing a high level overview of the intended functionalities of the system. Requirements for avoiding, detecting, and controlling faults are developed in this phase. Safe states need to be defined, into which the system should enter in the event of an erroneous situation. These stated requirements need to be implemented later in the system architecture or get implemented by external measures. The main work products out of this phase will allow the beginning of the product development at the system, hardware and software levels.

2.2.4. Part 4 - Product Development at System Level

Part 4 - Product Development at System Level, along with Parts 5 and 6 of the ISO 26262 follow a general V-model framework, starting with the requirements of the system, design, implementation, integration at different levels, verification and validation. This section deals with the top level system requirements and functional safety goals. There are three major clauses in this part -

- **Technical Safety Concept** includes the prerequisites for hardware and software development. Based on the identified high level functional safety goals, a necessary conversion to technical safety requirements need to be performed. This phase involves detailing system functionalities and

architecture, inter-dependencies among various sub-systems, specifying interfaces, and allocating resources for hardware and software components. Additionally, this phase requires the precise specification of safety mechanisms, including fault detection strategies and intervals, along with the corresponding transitions to safe states.

It is also necessary that different safety analyses are carried out in order to establish strategies for avoiding random hardware failures. Typically used techniques are Failure Mode Effect and Diagnostic Analysis (FMEDA), Fault Tree Analysis (FTA) and Dependent Failure Analysis (DFA). The primary aim of these analysis techniques is to identify faults and their causes and calculate the probability of failures of different hardware components, also known as *Failure Rates*. The end product or the realization of these measures will be evident during the stage of Product Development at Hardware Level, when the eventual metrics will be calculated. However, an initial assessment of the safety mechanisms could be defined by the **Diagnostic Coverage (DC)** at this stage. Diagnostic Coverage is the property of a Safety Mechanism to detect faults, and is quantitatively expressed in terms of percentage of faults detected. Necessary requirements for DC must be defined such that the failure rates are reduced to acceptable levels.

- **System and item integration and testing** involve the integration of sub-systems at various levels and the assessment of their outcomes, culminating at the top-level system. This process can only be performed once adequate development at lower levels (hardware and software) are completed. The primary objective of this clause is to make sure that the safety mechanisms are correctly implemented and integrated such that they fulfill all the required safety requirements at the system level.
- **Safety Validation** has to make sure that the safety goals have actually been achieved in the automotive, and that the product can now be deployed to be released, produced and installed in vehicles.

2.2.5. Part 5 - Product Development at Hardware Level

Hardware development refers to the process of developing the hardware of electrical and electronic systems. Part 5 - Product Development at Hardware level runs concurrently with software development, after a "Technical Safety Concept" has been formulated at the system level. In the context of this thesis, this section is highly significant as the objective is to verify functional safety at the hardware level. Therefore, it is crucial to understand the work products generated in this phase. The individual clauses in this part are discussed briefly as follows:

- **General topics**: All the necessary activities and processes required to develop hardware are discussed briefly in this clause. As an overview of the hardware development, it must include the hardware implementation of the Technical Safety Concept, analysis of hardware faults and their effects (both quantitatively and qualitatively), and the integration with the software development. Figure 2.5 shows an overview of the development process flow at the hardware level and its connections with other clauses in the ISO standard.
- **Specification of hardware safety requirements**: The main objectives of this clause are to specify the hardware safety requirements, which are derived from the technical safety concept and architecture specifications of the system design. The relevant properties and functionalities of the safety mechanisms and their ability to detect internal/external failures need to be well defined. It is also important to specify and refine the hardware-software interface (HSI) specification at this stage.

  These requirements and specifications must be verified with different criteria(which could include environmental conditions such as temperature, vibration or specific operational environment such as supply voltage) to ensure the consistency with the technical safety concept and system architectural design specification. This level of specification should also include attributes to ensure that the safety mechanisms are indeed effective and achieve the required coverage needed for the corresponding ASIL classification.

  Additionally, the requirements should comply with the fault tolerant time interval or with the maximum fault handling time interval. In view of ISO 26262, there are certain timing aspect considerations which need to be performed as well. Safety mechanisms will be completely evaluated

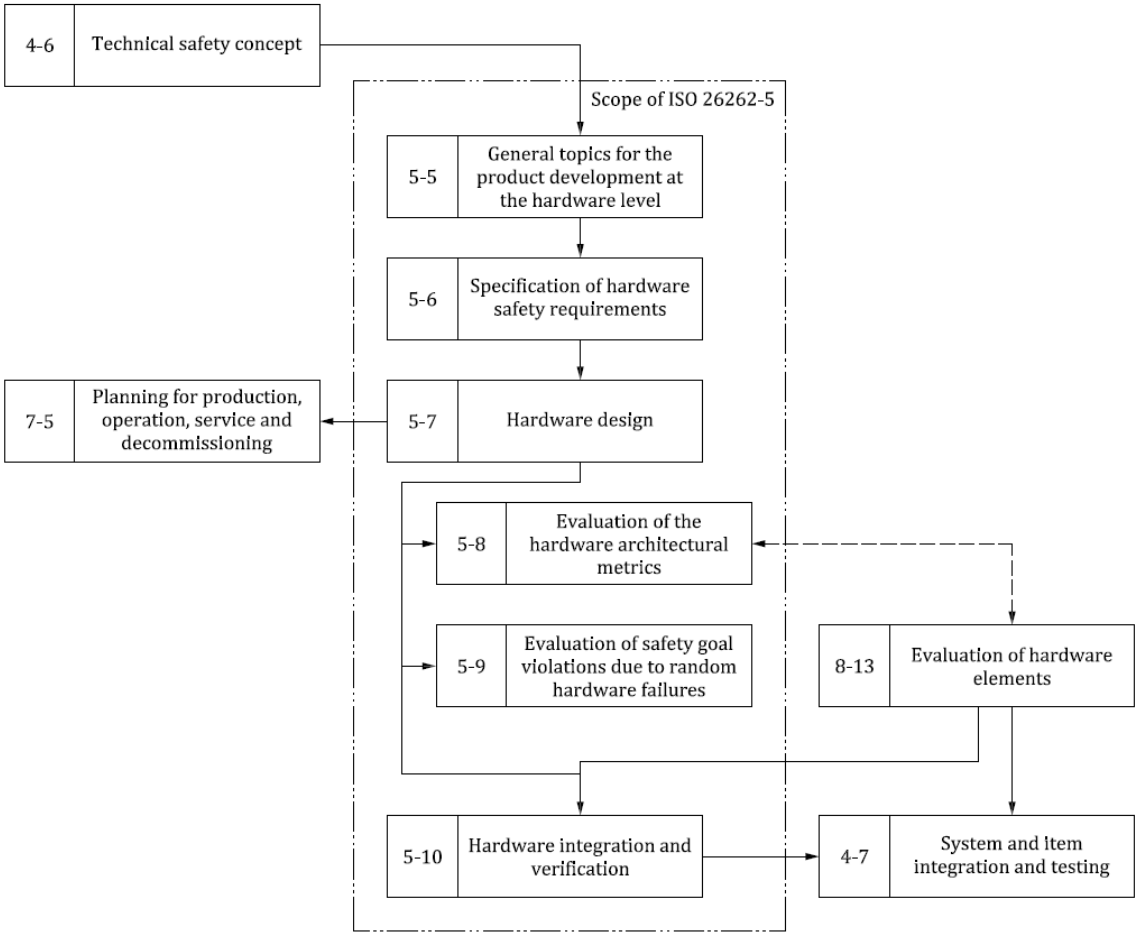


Figure 2.5: Product development at hardware level [18]

only when timing constraints are taken into consideration. The system should be able to detect faults and transition to a safe state within a specific time interval, also known as ***FTTI - Fault Tolerant Time Interval***. Figure 2.6 shows an illustration of FTTI, which also includes the time for the Safety Mechanism to detect the fault - Diagnostic Test Interval, along with the Fault Reaction Time and the corresponding transition to the Safe state. This timing consideration is important to be taken care of while verifying the hardware design.

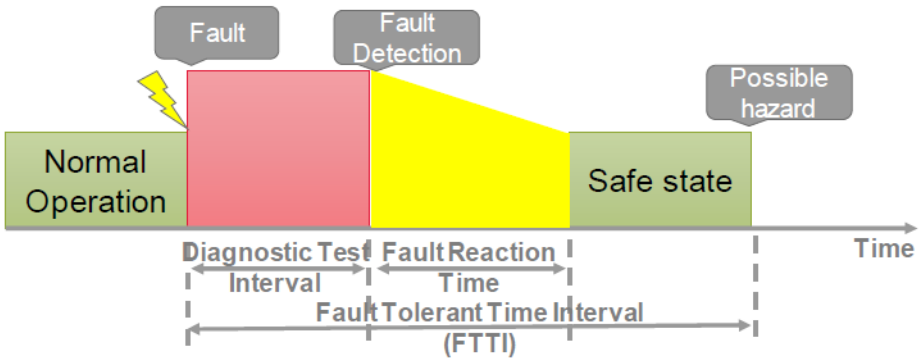


Figure 2.6: Fault tolerant Time Interval and Diagnostic Test Interval [19]

- **Hardware Design**: This is the next stage where a concrete hardware design is implemented.

In addition to ensuring that the functional safety requirements are met, it is also important to make sure that the actual functionality of the hardware is not affected while implementing the design. Necessary hardware architecture needs to be defined in order to implement the hardware safety requirements. These will be allocated to the hardware elements, which will be developed in compliance with the highest ASIL of any of the allocated requirements.

This clause defines the necessary properties of hardware design in terms of the ASIL levels. In general, components with higher ASIL levels need to have some essential properties such as precisely defined interfaces of safety-related hardware, maintainability and testability among others. In order to ensure that the implemented design is in conjunction with the safety requirements, this clause also extends the concepts of safety analyses. The hardware design must be created in a way that it supports all the necessary safety-oriented analysis and also considers the corresponding results.

For every safety related hardware, the safety analyses shall identify the following faults:

1. **Safe Faults**: Occurrence of such faults do not increase the likelihood of a safety goal being violated.
2. **Single-point faults/Residual faults**:
   a. *Single Point Faults*: Hardware faults in an element that leads directly to the violation of a safety goal and no fault in that element is covered by any safety mechanism.
   b. *Residual Faults*: Part of a random hardware fault which leads to the violation of a safety goal, occurring in a hardware element, where that portion of the random hardware fault is not controlled by a safety mechanism.
3. **Multiple-point faults (either detected, perceived or latent)**:
   a. *Detected fault*: Fault whose presence is detected within a prescribed time by a safety mechanism.
   b. *Perceived fault*: Fault that may be perceived indirectly (through deviating behavior on vehicle level).
   c. *Latent fault*: Multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver within the multiple-point fault detection time interval.

The above classifications are also illustrated in Figure 2.7, as will be discussed later during Fault Classification and Analysis. Such a safety analysis is imperative to demonstrate the effectiveness of implemented safety mechanisms in averting faults from causing various failures. After the completion of design phase, the design verification plan must be formulated and executed. This plan aims to verify the intended functionality, a process akin to the typical functional verification of chips. To summarise this clause, the main work products coming out of this phase are the hardware design specification, hardware safety analysis report, hardware design verification report, and finally the specification of requirements related to the production, operation, service and decommissioning, which will be used in the later parts of the safety lifecycle.

- **Evaluation of the hardware architectural metrics**: The main aim of this clause is to provide evidence regarding the suitability of the hardware architectural design in terms of detection and control of safety-related hardware failures. There are two hardware architectural metrics defined by ISO 26262: **Single Point Fault Metric (SPFM)** and **Latent Point Fault Metric (LFM)**. SPFM quantifies the danger posed by faults not protected by the Safety Mechanisms. LFM, on the other hand, denotes the threat posed by faults which do not violate a safety goal directly, but could be a risk in the presence of another fault.

  In order to calculate the hardware architectural metrics, we need to understand the kind of faults we need to inject. With regards to ISO 26262, the hardware safety analysis must consider the classification of **Stuck-At-0 (SA0)**, **Stuck-At-1(SA1)**, **Single Event Upset (SEU)** and **Single Event Transient (SET) faults**.

  The idea is to inject all the aforementioned faults at different locations in the system, and classify them into the different classes, as mentioned in the previous clause. Typically, at this stage, an FMEDA analysis is carried out to tabulate all the Failure Modes (FM) of the hardware element

under consideration, and the failure rates or Failure in Time (FIT - number of expected failures per one billion hours of operation) of the FMs. The Safety Mechanisms associated with the FMs are identified and the corresponding Diagnostic Coverage is calculated. With the help of this, the residual FIT rates are calculated for each hardware component. The FIT Rate is denoted by λ, and the different FIT rates corresponding to the fault classes earlier defined are mentioned as follows:

- λ_{SPF}: Failure rate of Single Point Faults (which directly violate a safety goal and is unprotected by SM)
- λ_R: Failure rate of Residual Faults (which are undetected by SMs)
- λ_{MPF}: Failure rate of Multi-point faults (which could only violate a safety goal in combination with another faults, and is the summation of $\lambda_{MPF,DP}$ and $\lambda_{MPF,L}$)
- $\lambda_{MPF,DP}$: Failure rate of detected MPF (covered by SM)
- $\lambda_{MPF,L}$: Failure rate of Latent MPF (faults in an SM)
- λ_S: Failure rate of Safe faults (which do not violate a safety goal)

The sum of all these faults (λ) is given by:

$$\lambda = \lambda_{SPF} + \lambda_R + \lambda_{MPF} + \lambda_S, \text{ where } \lambda_{MPF} = \lambda_{MPF,DP} + \lambda_{MPF,L} \tag{2.1}$$

Once the failure rates of these classes are defined, we can formally describe the SPFM and LFM.

The **Single Point Fault Metric** or **SPFM** is characterized by the classes of faults which can violate safety goals. The major contributing factors for this metric are the Single Point Faults and Residual Faults, both of which has the potential to directly violate safety goals. This metric is defined by the equations shown below:

$$SPFM = 1 - \frac{\textit{Single Point Faults + Residual Faults}}{\textit{Total Safety Related Faults}}$$

$$SPFM = 1 - \frac{\lambda_{SPF} + \lambda_R}{\lambda} \tag{2.2}$$

The **Latent Fault Metric** or **LFM** denotes the classes of faults which could be a risk in the presence of another fault, and are not covered by Safety Mechanisms. To calculate this, it is necessary to determine the coverage while excluding the classes of Single Point Faults and Residual Faults. LFM is defined by the following equation:

$$LFM = 1 - \frac{\textit{Undetected Multi Point Faults}}{\textit{Safety Related Faults - (Residual + Single Point Faults)}}$$

$$LFM = 1 - \frac{\lambda_{MPF,L}}{\lambda - (\lambda_{SPF} + \lambda_R)} \tag{2.3}$$

The target values for the two metrics, SPFM and LFM, are defined based on the ASIL level of the hardware component, and are shown in Table 2.4.

We take an example of an FMEDA analysis on a hardware element to illustrate the calculation of these metrics. It is important to note that such analyses can vary from one project to another, and the example shown below in Table 2.3 is a specific representation of the same.

The different acronyms in the given table are listed below:

- ID: Identifier associated with a Failure Mode
- FIT: Failure in Time
- FMD: Failure Mode Distribution
- SG: Safety Goal
- SM? (SPF): Safety Mechanism with regard to Single Point Fault

Table 2.3: FMEDA example for calculating hardware metrics

	Failure Mode	ID	FIT	FMD	Violates SG?	SM? (SPF)	DC-SPF	RF	MPF	SM? (MPF)	DC-MPF,L	MPF,L
Memory array	Memory element becomes corrupted to give wrong output values	FM1	100	70%	Yes	SM1	90%	7				
				30%	No							
	Stored ECC value gets corrupted	FM2	10	100%	No				Yes	No	10%	9
Total			110					7				9

- DC-SPF: Diagnostic Coverage for SPF
- RF: Residual Fault
- MPF: Multi-Point Fault
- SM? (MPF): Safety Mechanism with regard to Multi-Point Fault
- DC-MPF,L: Diagnostic Coverage for Latent MPF
- MPF-L: Latent Multi-Point Faults

We have an example of a memory array with two associated Failure Modes, FM1 and FM2. A Failure Mode Distribution (FMD) is linked to these FMs, indicating the percentage of area relative to a design block that could cause a failure. FMD is typically obtained from different statistical sources and is used in the eventual calculation of the number of faults relevant to the violation of a safety goal. FM1, with a FIT rate of 100, directly violates a Safety Goal. To address this error, a Safety Mechanism (such as ECC) is implemented, providing a Diagnostic Coverage of 90%. Assuming there are no single point faults (meaning there are no faults which are not covered by a Safety Mechanism), the total number of residual faults is then calculated as follows:

$$RF = 100 \times 0.7 \times (1 - 0.9) = 7$$

This FM does not have any Multi-Point Faults and therefore, there are no other contributing faults associated with it. FM2 does not directly violate a Safety Goal, as it represents a fault within the Safety Mechanism itself. It contributes to a multi-point fault and does not have any secondary Safety Mechanism for protection. The Diagnostic Coverage for this Multi-Point Fault is 10%. Again assuming that there are no detected multi-point faults, the total number of multi-point faults is determined by:

$$MPF = MPF,L + MPF,DP = 10 \times 1 \times (1 - 0.1) + 0 = 9$$

Finally, the SPFM and LFM metrics are given by:

$$SPFM = 1 - \frac{7}{110} = 93.63\%$$

$$LFM = 1 - \frac{9}{110 - 7} = 91.26\%$$

Upon examining the values of the SPFM and LFM for this hardware element, along with the target values for a specific ASIL as presented in Table 2.4, this element qualifies for an ASIL B rating. Another metric, which will be elaborated on in the next clause, also contributes to the calculation, affirming the qualification of the element for an ASIL B rating.

- **Evaluation of the safety goal violation due to random hardware failures**: The primary objective of this clause is to ascertain that the residual risk of a safety goal violation caused by random hardware failures is diminished to acceptable levels. ISO 26262 puts forth two methods to assess whether such a risk is sufficiently low.

  The first method is the calculation of **Probabilistic Metric for Random Hardware Failures (PMHF)**, which denotes the average probability of failure per billion hours of operational time of the hardware design. The evaluation of PMHF must prove that the combined safety target violation of all hardware elements are below a certain threshold for a given ASIL level. However, unlike the SPFM and LFM, ISO 26262 does not provide any method to calculate this metric. The data associated with different elements could be taken from statistical sources or relevant field data to arrive at the final PMHF metric. It could be also derived from different safety standards, such as the IEC 61508, as shown below in Equation 2.4:

$$PMHF = \Sigma\lambda_{SPF} + \Sigma\lambda_{RF} + \Sigma\lambda_{MPF,L} \tag{2.4}$$

  Table 2.4 denotes the target values for the three metrics, SPFM, LFM and PMHF in order to attain a given ASIL rating. Considering the example discussed above, the derivation of PMHF, based on IEC 61508, is as follows:

$$PMHF = 7 + 9 = 16FIT$$

  This value corresponds to an ASIL rating of B (<100 FIT), as also seen earlier with the other two metrics. If any of the metrics has a higher ASIL level than the others, then the metric with the lowest ASIL qualification will be attributed to the entire hardware element.

Table 2.4: Target values for metrics based on ASIL level

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

  The second method is an extensive evaluation of the different possible causes of a safety goal violation. Individual analysis of different failures must be performed at the hardware level, with the results of such an evaluation providing evidence that the risk of failures from these fault classes are acceptable. All different types of faults - SPF, MPF and RF, must be considered during this evaluation along with their failure rates. The diagnostic coverage of safety mechanisms must be sufficient to rationalize and mitigate the associated risks to a tolerable level.

- **Hardware Integration and Verification**: The main activities described in this clause aim at the integration of different hardware elements and verifying the compliance of the overall hardware design in accordance with the appropriate ASIL. Various methods need to be followed in order to derive test cases for testing the integrated hardware, with examples of such methods including analysis of requirements, internal and external interfaces, environmental conditions and operational use cases. Hardware integration tests must be used to verify the completion and correctness of the implemented Safety Mechanisms by means of **functional testing**, **fault injection simulation** and **electrical testing**. The final work products from the clause include the hardware integration and specification requirements along with the results and reports.

2.2.6. Part 6 - Product Development at Software Level

This part discusses the functional safety requirements at the software level which arises from the need of developing error-free software as more and more automotives include increased proportions of software in their design. An important thing to note in this case is that, software cannot fail randomly like electronic or hardware components. Thus, software faults fall into the category of systematic faults and in order to have an error-free software, it is essential to avoid such faults through systematic development. This part is further divided into the following clauses:

- **General topics** gives an overview of the software development and its integration with the hardware development in parallel, with respect to a V-model, as mentioned earlier. Software development for automotive applications must conform to the ISO 26262 standard and adhere to the requirements of the safety lifecycle, distinguishing it from conventional development practices in other domains.
- **Software Safety requirements** must be derived from the technical safety requirements in detail for implementation in software. This includes defining self-testing and monitoring mechanisms for the operating system, basic software, and application software. The clause encompasses requirements for detecting, indicating, and controlling faults in all safety-related hardware. Additionally, it is crucial to define safe states and establish procedures for achieving them in the event of a failure. The specification also outlines necessary interface requirements between the software and hardware.
- The **software architecture** should implement all the functional requirements and safety mechanisms at the software level. It is imperative to conduct thorough safety analysis to prevent *dependent failures*, ensuring that error-prone software does not compromise the functional safety of critical software components. This analysis should encompass aspects such as runtime behavior, memory areas, and message traffic.
- **Unit design and implementation** is a typical part of every software development process. A software unit design is essential as it could also be used as a model for applying model-based software development.
- The subsequent phase involves **Unit verification**, which is needed for validating the accurate implementation of safety mechanisms through thorough testing. This process ensures that test coverage requirements are met and verifies the absence of unintended functionality in the code. Additionally, it ensures the availability of essential resources, including execution time, memory, and message throughput, to facilitate smooth code operation.
- After the development of individual software units, **software integration and verification** of these units need to be carried out to ensure the intended functionality of all functions at the software level. Questions raised in the previous clause must also be taken care of here at the overall software level.
- **Testing of embedded software** is essential to verify that the software meets safety requirements when deployed on specific hardware in a given target environment. To achieve this, the software must undergo testing in various environments. First, the testing should be done with the help of simulation. Second, it should be tested by placing the software in a network of real electronic control units. Finally, it should be tested in a real-life scenario by placing the system in a prototype vehicle.

To summarise, software development in ISO 26262 takes a step further than the usual route by ensuring that all safety requirements are met with the help of safety mechanisms implemented in software, and that there is no unintended functionality. Systematic faults must be avoided at all costs with proper integration and testing methods and test coverage must be measured to evaluate the completeness of testing and achievement of test goals.

In the context of the thesis, Parts 1-6 cover majority of the essential background knowledge needed for Functional Safety Verification in automotive chips. The subsequent sections of the standard, spanning from 7 to 12, are briefly discussed to provide a general idea about the topics.

2.2.7. Parts 7 - 12

Part 7 - Production, operation, service and decommissioning deals with the activities in the lifecycle after the development has been completed. It is essential to verify that the safe production and installation of electronics is carried out. Workshops conducting repairs must avoid any safety hazards, and adherence to ISO 26262 is therefore necessary for proper planning. The field observation process aims to examine defective parts, analyzing deviations from safety concepts and checking if software updates or hardware replacements are needed. These steps conclude the vehicle's lifecycle considerations.

Part 8 - Supporting Processes provides guidelines and additional considerations to assist engineers in achieving compliance with the safety lifecycle. There are clauses with defined objectives, prerequisites and expected work-products, describing critical processes that support the entire lifecycle. They address aspects such as interfaces in distributed developments, specification and management of safety requirements, configuration and change management, verification processes, documentation management, and ensuring confidence in the use of software tools, which will also be discussed in detail in Section 2.4. Additionally, the clauses provide information on topics like the qualification of software components, evaluation of hardware elements, proven in-use arguments for existing elements to be reused, interfacing with applications outside the scope of ISO 26262, and integrating safety-related systems not developed according to the standard.

Part 9 - ASIL-oriented and safety-oriented analyses must be carried out in conjunction with the development process throughout the lifecycle. These analyses are essential for gaining a precise understanding of the origins and effects of different kinds of faults, thereby ensuring a robust and functionally safe design.

Part 10 - Guidelines on ISO 26262 provides in-depth explanations and further recommendations for better understanding. This part is more of an informative section, but still quite helpful.

Part 11 - Guidelines on application of ISO 26262 to semiconductors is one of the two additional parts introduced in the ISO 26262 version of 2018. This part offers additional insights into understanding Safety Mechanisms and the reliability of semiconductors within the functional safety process. Many concepts mentioned briefly in preceding sections are further explained and explored in greater detail within this part.

Detailed explanations are provided for the quantification of transient faults, the calculation of failure rates sourced from industry data, and examples illustrating Diagnostic Coverage in different systems. The DFA section delves into the various causes and initiators of common cause and dependent failures. The recommended workflow and subsequent preventive measures for addressing these failures are also thoroughly discussed. Further definitions and guidance regarding fault models for components, including memories, failure modes of common digital blocks, and the analysis and estimation of transient behavior and diagnostic coverage, are documented in detail. Similar analysis is also followed for analog and mixed signal components, programmable logic devices, multi-cores and sensors/transducers, covering majority of the semiconductor technologies for consideration under functional safety development.

Part 12 - Adaptation of ISO 26262 for motorcycles has been specifically added to the standard for explaining the functional safety lifecycle and its adjustment for motorcycles. Majority of the concepts still remain the same, including the HARA process, with some minor tweaks in the clauses relating specifically to motorcycles. For example, additional terminologies have been added to the vocabulary, and ASIL has been substituted with Motorcycle Safety Integrity Level (MSIL). The MSIL levels translate to one level lower than the ASIL levels (i.e. MSIL B is handled according to the rules of ASIL A), and the classification table remains almost identical. The different rules already discussed in the previous parts of this standard are then modified according to these updated classifications.

This concludes an overview of the Functional Safety concepts in relation to ISO 26262 standard. The subsequent section explores various fault classification techniques and the corresponding analysis for evaluating the final outcomes of hardware metrics.

2.3. Fault Space Analysis and Classification

ASIL ratings of automotive components determine the quality and robustness of the overall system. In order to determine ASIL levels, it is important to understand the effects of faults in these components and their corresponding classifications. Thus, the next logical step is to analyse different fault effects and determine their classifications, along with the failure rates, to calculate the required hardware architectural metrics.

There are different ways in which fault analysis can be performed, aided by different technologies. In general, the main aim is to classify the entire fault space into the classes described by ISO 26262 and as shown in Figure 2.7. Any hardware element under consideration is divided into various failure modes, and the corresponding coverage of the safety mechanisms associated with these FMs are identified to classify the various faults. Once the fault classes are known, we can calculate the necessary metrics and qualify the element to the appropriate ASIL level.

Various methods can be employed for fault classifications, as will also be elaborated in Section 3.2.

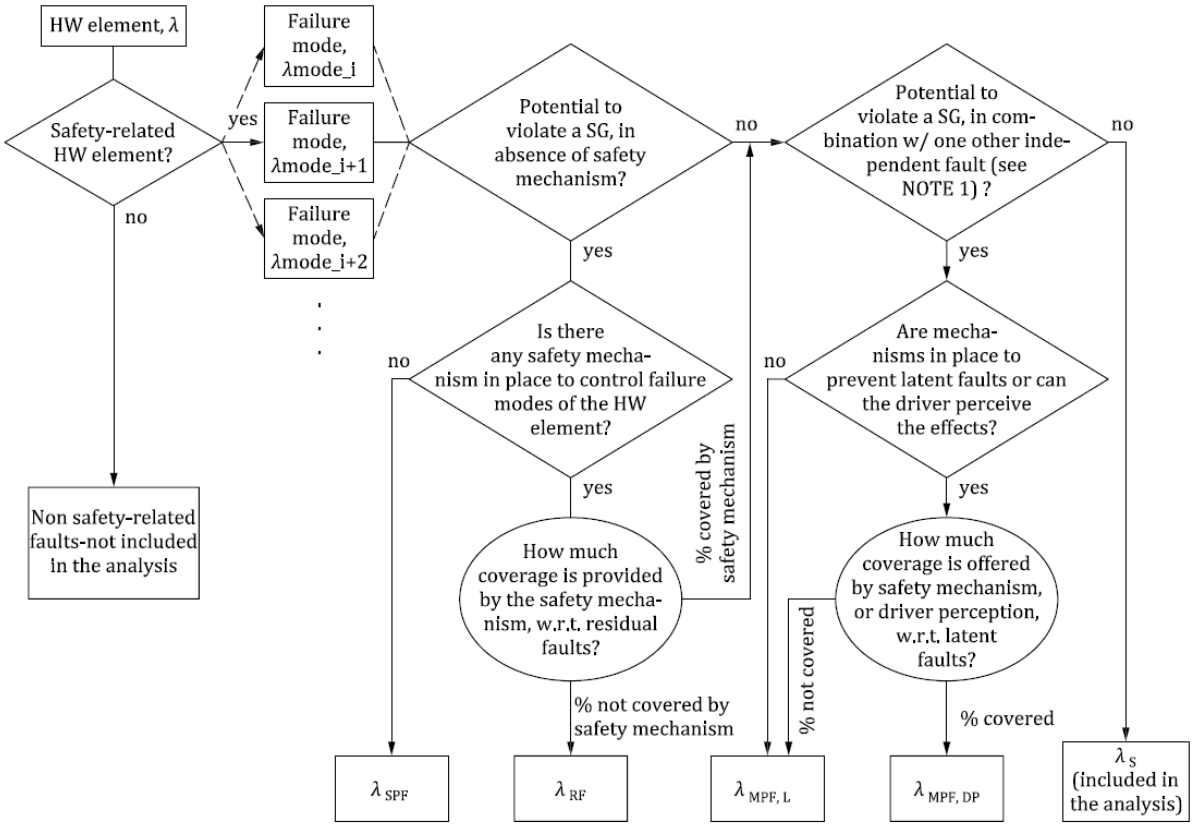


Figure 2.7: Flow diagram for failure mode classification [18]

However, for a concise overview of the technologies and methodologies in use, a preliminary discussion is provided in this section.

The primary methodology recommended by ISO 26262 is **fault injection simulation**, wherein different faults are injected at various design points and subsequently simulated using functional tests. Comparisons between the design outputs and those of a good (fault-free) simulation determine the fault classification. This approach is supported by various EDA tools, simplifying the process and minimizing manual effort. It represents an improvement over **testbench-based fault simulation** methods [20][8], which manually introduce faults and force corresponding signals during test simulation — a process which requires a lot of manual work and lacks scalability for larger and complex designs. Other classification methods include **radiation-based testing** [21] for transient fault analysis, primarily applicable post-product development, and not suitable for RTL-level testing. While **fault emulation** methods are feasible and emulation platforms are available for fault injection, their discussion is not presented here as fault simulation is the suggested ISO 26262 qualification methodology. On the other hand, **formal methods** offer an extensive means of identifying safe faults through different analysis techniques, requiring minimal user intervention. Identifying safe faults early in the lifecycle saves critical simulation time, emphasizing the importance of formal methods. Considering the feasibility of fault injection simulation and formal methods with respect to the research problem at hand, this section delves deeper into these two methods.

- **Fault Injection Simulation**: This is the methodology suggested by ISO 26262 for fault space classification and is widely used in different solutions. There are dedicated tools provided by leading EDA vendors for fault simulation purposes. These tools can analyze Register Transfer Level (RTL) or Gate Level Netlist (GLN) design descriptions to simulate their behaviour. The fault class is determined by comparing the outputs of the Design-under-Test (DUT) with and without faulty behaviour. The main steps involved in the fault injection simulation flow are as follows:

1. **Compilation/Elaboration of RTL/GLN designs**: The given design has to be compiled with the necessary switches and arguments required to prepare for fault simulation.
2. **Generation of fault list and optimisation**: This is where the fault space is defined and generated. Users can define the targets for fault generation, the type of fault models to be used (Stuck-At, Transient or any other variations supported by the tool), fault injection times. Different types of faults could also be excluded in this stage from being generated, thus saving valuable time in not instrumenting those faults.
3. **Good simulation**: The DUT is simulated without any injection of faults to generate a golden database for reference. Users typically define specific signals in the design to act as observation and checker points. Observation points capture information related to functional outputs that directly impact the design output, while checker points are integral to the Safety Mechanisms and could be regarded as signals for fault detection or alarms. These user-defined signals are recorded during a fault-free simulation run and later compared during fault simulation.
4. **Fault simulation**: Faults are injected one by one into different locations and the DUT is simulated under the influence of these faults. The observation and checker points are once again recorded during this faulty run in order to find out what the classification of the faults are.
5. **Fault classification and reporting**: The primary idea behind Safety Mechanisms is the fact that it should be able to detect faults in the design. Under an ideal case with a perfect Safety Mechanism, all faults would be detected at the checker outputs, which would lead to achieving complete coverage. However, faults could also be detected at the functional outputs, and not at the checker outputs, which is exactly the kind of scenario we want to avoid. Based on where the faults are observed/detected, classifications are made accordingly. Different tools provide varied classifications, but the basic idea still remains the same.

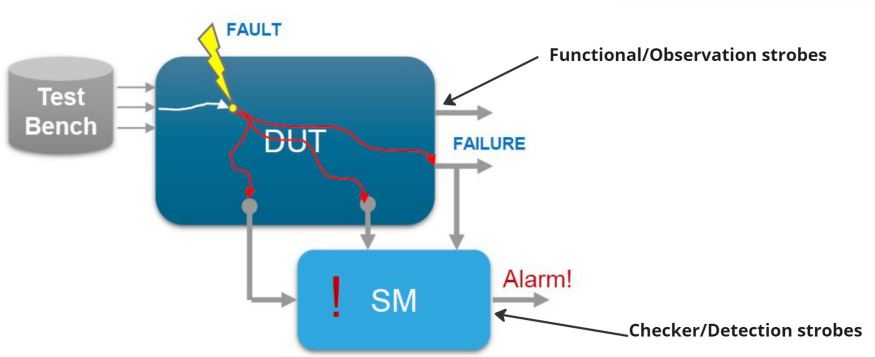


Figure 2.8: Design with Safety Mechanism(configured with Observation and Checker strobes)

Let us consider a DUT with a Safety Mechanism to detect faults and erroneous situations, as shown in Figure 2.8. In this setup, one or more functional outputs of the DUT are designated as **functional/observation strobes**, while a corresponding set of outputs from the Safety Mechanism logic serves as **checker/detection strobes**. If a fault introduced at any point in the design propagates to a functional strobe, it is labeled as **Observed**; if not, it is deemed **Not Observed**. Conversely, if the fault reaches a checker output, it is categorized as **Diagnosed**; otherwise, it is marked as **Not Diagnosed**.

The final classification of a fault occurs based on its propagation to both these strobes. When a fault is injected and propagates to both the functional and checker strobes, it is termed as an Observed Diagnosed Fault (OD). If a fault is injected and propagates solely to the checker strobe, without affecting the functional strobe, it is labeled as a Not Observed Diagnosed fault (ND). The first letter in the two-letter classification refers to the fault detection at the functional strobe, whereas the second letter corresponds to the detection at the checker strobe. Table 2.5 shows the combination of these classifications with one another, along with the acronyms, which

will be used for the remainder of this report, if and when applicable. These are the most basic classifications which will be seen in any functional safety verification setup, with the classification names varying from tool to tool. Generally, there is a subset of Safe (S) faults as well, faults which cannot be propagated to any output **regardless of the inputs provided to the design (fully exhaustive input space)**. Not Observed Not Diagnosed faults (ND) also do not propagate to either of the outputs, but only for the set of tests that we provide. However, there is a possibility that a certain set of inputs might propagate this fault to one of the outputs, and hence is not considered a true Safe fault. Further, there are more fault classifications provided by different tools, based on different situations observed in the design and test cases.

Table 2.5: Fault classifications

	Detected Functional point	**Undetected Functional point**
Detected Checker point	Observed Diagnosed (OD)	Not observed Diagnosed (ND)
Undetected Checker point	Observed Not Diagnosed (ON)	Not Observed Not Diagnosed (NN)

Based on the above classifications, we can make an estimation of the Diagnostic Coverage of the Safety Mechanism, which is essentially the percentage of faults detected by an SM. The DC is then given by the equation:

$$\text{Diagnostic Coverage (DC)} = \frac{OD + ND}{OD + ND + ON} \times 100\% \tag{2.5}$$

By examining the equation, it becomes evident that the enhancement of Diagnostic Coverage is achievable through the reduction of faults observed at functional points but not detected at checker points (ON). This stands as a key motivation behind the Functional Safety Verification process. Expanding the fault classifications introduces additional classes to the DC equation and accordingly, fault analysis must be done to increase the DC.

- **Formal methods**: A crucial initial step in Functional Safety verification involves identifying Safe faults, namely faults that do not influence the design outputs. Once these faults are recognized, there is no need to inject them for simulation, resulting in significant time savings. Labeling a fault as Safe is dependent upon proving its untestability, implying that no available combination of test stimuli can propagate the fault. In this regard, formal methods emerge as a robust alternative, as they are not confined to specific times or states. Instead, their scope is global, encompassing every evaluation context and test stimuli for an extensive assessment.

  Various EDA vendors incorporate fault analysis capabilities into their formal solutions. In broad terms, these solutions automatically generate properties that validate the behavior of faulty designs, eliminating the need for expertise in formal languages. Furthermore, they facilitate integration with Fault Injection Simulators, optimizing fault lists and streamlining simulation campaigns. Tools employed for formal analysis typically employ two primary fault analysis techniques: Structural Analysis and Formal Analysis.

  - **Structural Analysis**: This kind of analysis is done with the help of the physical characteristics of the design. There are several techniques under structural analysis which are utilised to find out Safe faults, briefly explained below:

    1. **Cone of Influence (COI) analysis**: The primary idea behind this technique is to identify signals and points in the design which directly come in the cone of influence of the strobing points. An example of this is illustrated in Figure 2.9. Considering that o_0 is the only strobe of this design and looking at the cone of influence for this point, a fault at i_4/s_2 or i_5/s_3 will not cause a failure, and hence these faults can be considered Safe. All the remaining faults need to analysed further to decide whether they are safe or not.
    2. **Constant analysis**: This technique also goes by different names such as **Controllability analysis** and **Activation analysis** in different tools. If any signal is held at a constant value, then the corresponding fault at that constant value can be considered

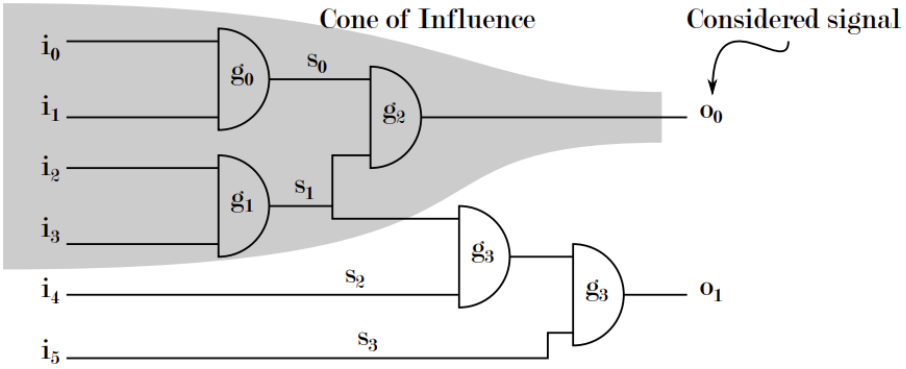


Figure 2.9: Cone of influence example [22]

Safe. Considering the example shown in Figure 2.10, if the input B stays at 0 the entire time, then the faults SA0 and SA1 at the shown locations are not controllable, and hence Safe.

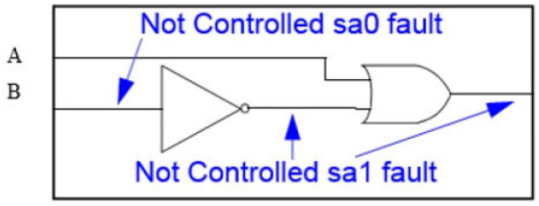


Figure 2.10: Controllability analysis [23]

3. **Propagation/Observability Analysis**: This analysis indicates the faults which cannot be observed at the strobes and occur when the workload obstructs the propagation of the fault from a specific fault location to an observation point. These faults manifest when a dominant signal to a gate remains static and does not toggle, persisting at either 0 or 1 throughout, depending on the gate. Let us take the example of a simple NAND gate where one of the inputs, B, remains 0 throughout. Then, the SA0 and SA1 faults at A will be blocked, as shown in Figure 2.11.

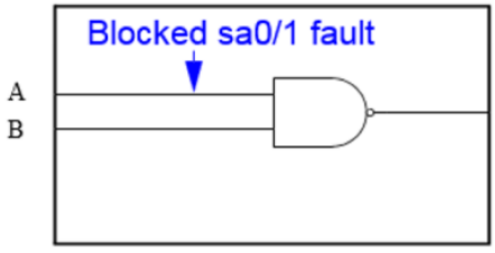


Figure 2.11: Propagation analysis [23]

These are some of the basic structural analysis techniques seen in formal tools. Nevertheless, various tools offer more sophisticated features to identify additional Safe faults. Direct examination of inputs is possible to determine if a fault can be activated or if it will never propagate to the functional output. Furthermore, checks can be conducted to see if faults can consistently be detected at the checker outputs or if a propagated fault will always be detected. These kind of improved techniques help in further pruning the fault space required for fault simulation.

– **Formal Analysis**: Formal verification utilizes mathematical analysis to navigate the design state space exhaustively, ensuring the accuracy of the design. It complements simulation based verification methods. The general idea behind formal analysis in functional safety verification lies in two replicas of the same design being generated by the tool - one for the normal (good) state and another for the faulty (bad) state, as shown in Figure 2.12. First, the tool administers identical inputs to both machines and incorporates monitors at observation points. Discrepancies in the observation points between the two machines signify the propagation of a fault. Thus, by inducing the fault effect in the faulty machine, we can discern the fault subclass. This iterative process is replicated for all elements within the fault space.

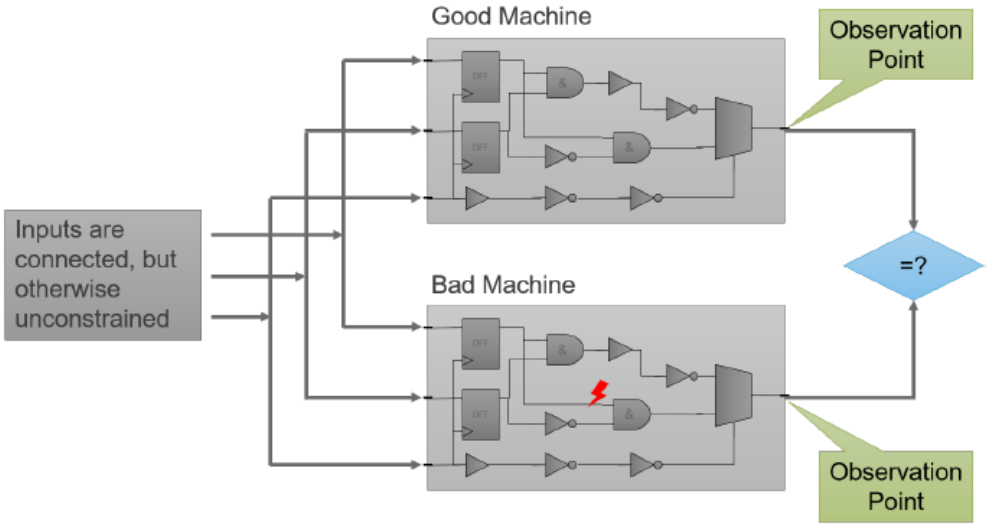


Figure 2.12: Formal analysis [24]

Formal methods depend on property verification to identify different fault classes. These tools automatically generate properties to assess the activation and propagation of faults. Activation analysis checks whether any combination of inputs can activate a fault at the functional outputs. Therefore, the formal engine must verify a property affirming that the fault target can adopt a logic value opposite to the fault model, allowing for the activation of such a fault. The formal engine must validate these properties for every possible combination of input values. If a property is proven false, the activation of the corresponding fault is deemed impossible, rendering it a Safe fault.

The formal analysis capabilities of these tools can provide additional insights into faults. For instance, upon confirming a property that verifies the propagation of a fault, the tool can give the combination of input stimuli validating that property. Therefore, we can obtain a counter-example for each proven property. Moreover, replicating such input stimuli in the simulation environment can enhance Fault Injection simulation results.

2.4. Tool Qualification

Given that we are engaging with EDA tools for functional safety verification, we need to understand the impact of these tools on the development process. ISO 26262 offers detailed guidelines on the qualification of software tools, with the main goal of presenting evidence that the tool is appropriate for use in the functional safety development process. It is essential to document and analyze the use cases for a tool, thus aiming to assess whether a malfunctioning software tool or its erroneous output could result in the violation of a safety requirement. Further, the analysis should evaluate the likelihood of preventing or detecting errors in the tool's output. The outcome of this analysis contributes to determining the required **Tool Confidence Level (TCL)**.

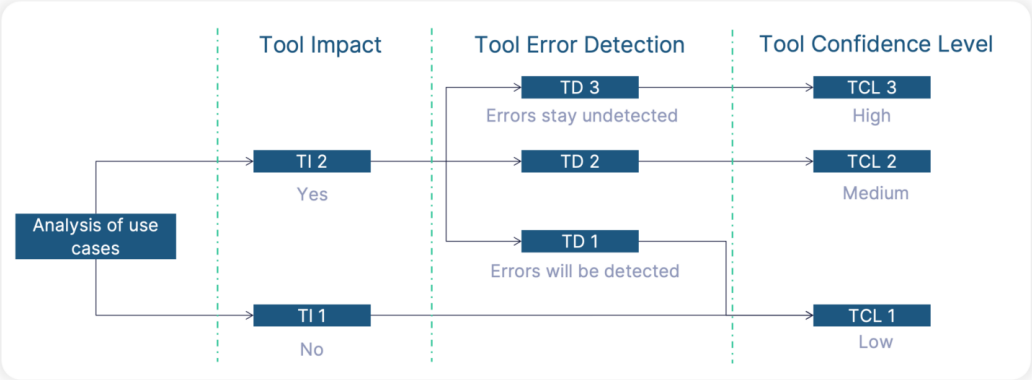


Figure 2.13: Tool Confidence Level Classification [25]

The process of determining the TCL of a given tool is illustrated in Figure 2.13. To initiate the process, we need to first assess the **Tool Impact**, which is ascertained in two levels, TI1 and TI2. If a tool has the potential to introduce errors or even fails to detect errors, it significantly influences the ultimate product quality, resulting in a TI2 classification. Conversely, if a tool plays no role in determining the final product quality, it falls under the category of TI1. For instance, let us take the example of a code generator or a compiler. Given that these tools generate code that are integrated into the final product, a malfunction in such tools can indeed impact the product quality, resulting in a TI2 classification.

Following the assessment of Tool Impact, the next step involves determining the **Tool Error Detection** (TD) level to assign the Tool Confidence Level (TCL). In the case of a TI1, the default classification is TCL1, and further TD evaluation is unnecessary. However, for tools with a high impact, it is important to determine whether the introduced errors can be detected by the tool and the probability of such detection. If there is a very high probability of detection (TD1), the tool is designated TCL1. In instances with a moderate probability of error detection (TD2), a TCL2 is assigned. Finally, if there is a minimal probability of detection (TD3), the tool receives a TCL3 classification. For example, considering the code generator/compiler scenario, if tests are not conducted to validate the generated code, the likelihood of error detection is very low, resulting in a TD3 classification and ultimately a TCL3. The exact probability metrics are not defined by the standard and is dependent on the functional safety engineers in the planning process.

TCL1 is the lowest tool confidence level and implies that the tool has no particular say in the final quality of the product. For this classification, a tool qualification is not necessary as well. **When it comes to EDA tools used for fault simulation purposes, their usage does not directly impact the quality of the product design, as these tools do not introduce errors into the product themselves. They are therefore assigned a TCL1 classification.** TCL2 and TCL3 corresponds to medium/high confidence level tools, meaning they have significant impact on the quality of the product. Thus, a tool qualification needs to be performed to ensure the reliability of the tool.

Table 2.6: Suggested Tool qualification methods

Method	ASIL A/B/C/D
Increased confidence from use	++
Evaluation of the development process	++
Validation of the software tool	+
Development in compliance with a safety standard	+

(+ - method recommended; ++ - method highly recommended)

Tool qualification is primarily the user's responsibility within a specific project context, although tool providers can facilitate this process. As shown in Table 2.6, there are four suggested methods

provided by ISO 26262:

- *Increased Confidence from Use* relies on the tool's successful use in a previous project.
- *Evaluation of the Development Process* involves a detailed analysis of the tool development and is typically pre-qualified by an authority.
- Another approach is *Validation of the Software Tool*, requiring the development of a comprehensive test suite covering all tool use cases, which can be undertaken by either the user or the tool vendor.
- Lastly, aligning tool development with a safety standard is essential in order to demonstrate the reliability of the tools being used.

Having laid down the necessary concepts for functional safety verification in this chapter, we now present the literature review and explore the current state-of-the-art in this field in the next chapter.

3

State-of-the-art

The primary research question in this thesis involves identifying potential discrepancies in the classification results of fault simulation EDA tools. If such discrepancies exist, what can be done to address them and develop a robust and dependable verification methodology? Based on results, safety enhancements must also be proposed for designs to improve fault detection and recovery mechanisms. To address these questions, we investigated three primary themes aimed at identifying current trends and technologies in Functional Safety (FuSa). The **first theme** involved obtaining an overview of Functional Safety concepts and ISO 26262, along with exploring general trends in functional safety topics to better understand the concept. The **second theme** delved into the trend of verification methodologies employed in functional safety solutions, with the goal of gaining insights to develop a solution for the research problem. As shown in Figure 3.1, we looked at approaches developed specifically with FuSa EDA tools, along with optimisations built on top of it. We also looked at techniques such as emulation, radiation based testing and testbench based fault injection in a different category. The **third theme** focused on understanding the safety mechanisms typically used in functional safety solutions, either based on error prevention/detection or error correction. A complete overview of the literature review conducted is presented in Figure 3.1, with each of the individual themes discussed in detail in subsequent sections. We conclude with summarizing the key takeaways obtained from the literature review, reiterating the research objectives, and paving the way for the development of proposed solutions.

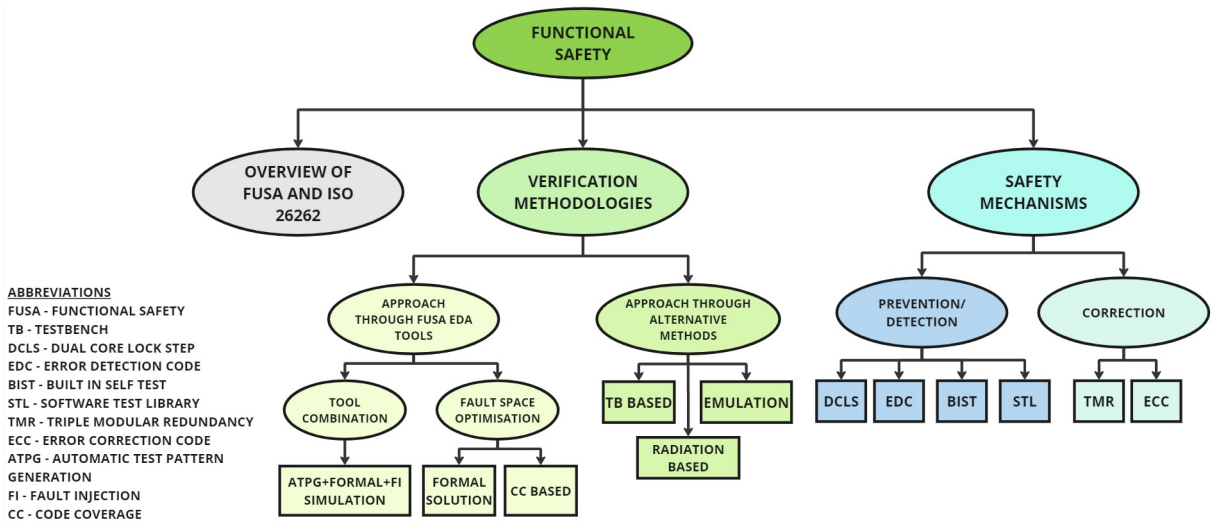


Figure 3.1: Overview of literature review topics

3.1. Overview and Functional Safety Analysis

Safety analysis techniques play a pivotal role in the functional safety lifecycle, offering a comprehensive insight into design, failure modes, safety mechanisms, and failure rates. In [19], an overview of safety analysis techniques is presented, covering both qualitative and quantitative methods. Quantitative approaches like FMEDA and Timing Analysis are widely adopted in functional safety solutions. Additionally, qualitative techniques such as DFA list the causes of failures not addressed by quantitative methods, focusing on common cause and dependent failures. Common reasons for such failures stem from random hardware faults in shared resources such as clocks, power supplies, or reset logic. Countermeasures involve independent monitoring of shared resources, diversification of impact (e.g., clock delays between master and checker cores), physical separation, among others. [19] also provides a concise overview of safety mechanisms, including Built-in Self Test (BIST), Triple Modular Redundancy, and Dual Core Lock-Step (DCLS) architectures. Various BIST variations are employed to enhance both SPFM and LFM, addressing online and offline BIST scenarios.

In [14], the authors discuss challenges of EDA tools in functional safety applications and outline their requirements in the lifecycle. The bare minimum requirements involve automating critical tasks and providing users with a coherent database for better design understanding. The paper also analyses fault injection strategies, risk assessment, and failure analysis of a Lane Keeping/Departure system, offering practical insights into the process. [20] discusses the requirements for fault injection environments, emphasizing factors like scalability, re-usability, rapid simulations, and multi-user control. The discussion provides guidance on selecting EDA tools based on whether they fulfill these specified requirements. Further, [26] provides insights into diagnostic coverage and associated overheads for commonly used safety mechanisms, as summarized in Table 3.1. [27] further highlights diagnostic measures related to memory elements - RAM test (checkerboard/march) and single bit redundancy, which provide low diagnostic coverages, and Error Detection-Correction Codes (EDC) and block replication, which have high diagnostic coverages. A general overview of these diagnostic measures serves as a quick reference guide to understanding what safety mechanism bodes well in what kind of a system.

Table 3.1: Diagnostic coverage of Safety Mechanisms with relevant overheads [26]

Safety Mechanism	Transient fault DC	Permanent fault DC	Area overhead	MIPS overhead
Lock-step CPU	High	High	High	Low
Hardware self-test for CPU	NA	High	Medium	Medium
Software self-test for CPU	NA	Medium	Low	High
Parity for memories	Low	Low	Low	Low
ECC for memories	High	High	Medium	Low
Self test for memories	NA	High	Medium	Medium

[28] outlines a general functional safety verification process aligned with Part 5 of ISO 26262. The example focuses on a safety microprocessor equipped with safety mechanisms like dual modular redundancy, CRC, parity, and self-tests for hardware. The evaluation involves synthesizing the safety design, determining failure modes, assessing corresponding failure rates, and calculating the diagnostic coverage of safety mechanisms. Using an EDA tool, TetraMax (primarily an ATPG tool, not a dedicated fault simulator), the SPFM, LFM, and PMHF metrics are established and compared with ISO target values to determine the ASIL level. Notably, the approach lacks consideration for transient faults and does not include fault simulation, as suggested by ISO 26262 for fault testing. Despite these limitations, the approach sheds light on a commonly used process for hardware verification in accordance with ISO 26262.

3.2. Methodologies and Techniques for Functional Safety Verification

One of the main aims of the thesis is to develop a generic framework which can be used to perform functional safety verification on a generic hardware design. Different methodologies and techniques to

perform fault simulation are looked at in order to understand the state-of-the-art and understand the scope of improvement.

3.2.1. Verification approaches without dedicated FuSa EDA Tools

In this section, we explore verification methodologies for FuSa that have been devised without relying on specialized EDA tools for functional safety. Examples of such approaches include verification using Functional Testbench setups, Radiation-based testing and emulation.

In [11], a verification flow similar to the one outlined in Section 2.3 for fault injection simulation is adopted. The authors raise a critical issue in safety verification, which is typically conducted at the gate level to leverage translated physical features effectively with functional safety EDA tools. However, discovering bugs at the gate level would require subsequent changes to the RTL, prompting a repetitive process to achieve the desired diagnostic coverage and stability. This issue aligns with one of the key questions posed in our research problem—exploring the feasibility of using EDA tools for functional safety at the RTL stage.

In [20], the authors present a customized Design Verification Environment that incorporates fault injection campaigns and simulation. The EDA tools employed are only for simulation rather than fault injection and classification. A tool is devised to extract features from RTL/GLN for generating fault lists and pruning them based on static analysis, akin to formal methods. Additionally, the authors employ clustering techniques to condense the fault space by grouping similar elements (e.g., buses, hierarchical modules). Subsequently, only one fault from each cluster is injected during fault simulation. Fault injection is executed using force/release statements to synchronize injection times, with detection points set in the testbench to classify faults and ascertain detection times for timing constraints.

An FMEDA-based fault Injection and Data Analysis (FIDA) framework is discussed in [8], which supports the FMEDA analysis in conjunction with fault injection and simulation. First, the design is parsed to generate targets for fault injection. FIDA subsequently produces a set of testbenches to run fault-free and faulty simulations, yielding classification results. The framework then automatically generates corresponding metrics and the final FMEDA report, effectively reducing manual effort for the designer.

[21] introduces a distinctive approach to functional safety verification through the development of a System Validation Tool (SVT). Unlike conventional bottom-up methods like FMEDA, SVT adopts a top-down strategy to assess transient faults, particularly Single Event Effects (SEE). Employing radiation-based fault injection and automatic test generation, SVT applies significant stress on the hardware logic and memories of a System on Chip (SoC) designed for automotive applications. The tool generates targeted random test vectors to cover various testing scenarios, comparing the device register state to the expected state post-testing. Classification is based on detection and comparison values, eventually leading to the calculation of SPFM and LFM metrics. However, the study lacks comparisons with traditional testing methods to assess the tool's effectiveness in detecting otherwise undetected failures.

Emulation based fault injection is also seen in [29], wherein a customised fault injection framework is developed in hardware (contributing to additional overhead) and then emulated using a dedicated EDA emulator. Further, a lot of these emulation based fault injection techniques are often carried out using Field Programmable Gate Arrays (FPGAs), as seen in [30][31][32] among others. This is also not in line with our thesis research as we want to verify RTL designs without the overhead of an FPGA.

The techniques discussed above are based on tools and methods which are custom developed to perform fault injection and classification, without the usage of dedicated FuSa EDA tools. Now, we look at methodologies developed using EDA tools, including formal methods and alternative technologies like Automatic Test Pattern Generation (ATPG).

3.2.2. FuSa EDA tools-based verification methodologies

One of the focal points in functional safety verification using fault simulators include the identification of Safe faults, thereby contributing to **Fault Space Optimisation**. Identifying Safe faults early in the process helps save valuable time by avoiding fault simulation on targets that are already deemed safe. Formal tools play a significant role in this process, offering support by employing various strategies to isolate safe faults. Fault simulation tools themselves come equipped with capabilities to identify safe faults.

Different methodologies leverage diverse techniques to enhance the Tool Confidence Level in this

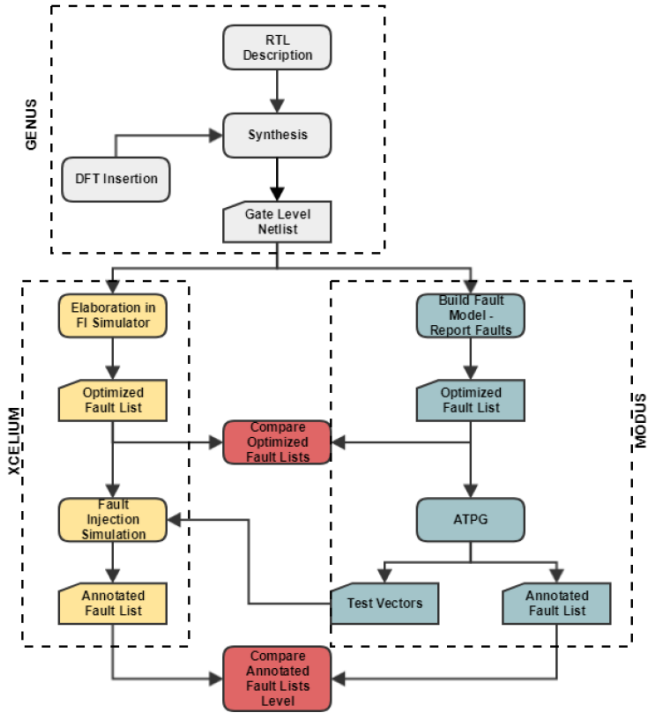


Figure 3.2: Combination of ATPG with fault injection simulator to compare fault lists [33]

regard. For instance, in [33], two parallel flows are executed using different technologies to compare fault lists generated by both tools. This process, illustrated in Figure 3.2, involves running a fault injection simulator concurrently with the ATPG flow to produce original and optimized fault lists. These fault lists are compared to check for any discrepancies and improve the TCL of the tools. Similarly, in [34], the formal tool JasperGold FSV is utilized to prune fault lists and compare results with optimized lists generated by a fault injection simulator, resulting in an average reduction of 29.5% in fault lists.

In [9], [10], a combined approach utilizing ATPG, formal, and fault injection simulators is used for functional safety verification. Illustrated in Figure 3.3, this method leverages formal and ATPG tools concurrently to generate optimized fault lists, followed by the utilization of ATPG-generated test vectors for fault simulation. At the conclusion of this process, annotated fault lists are compared, and equivalencies across tools (e.g., Safe in formal, Ignored in ATPG, and Untestable in Simulator) are mapped to produce final reports. Manual inspection is required for certain discrepancies to assign the fault's final classification. This method was tested across four different designs with Stuck-At faults at all cell ports, achieving a fault detection rate of at least 99%.

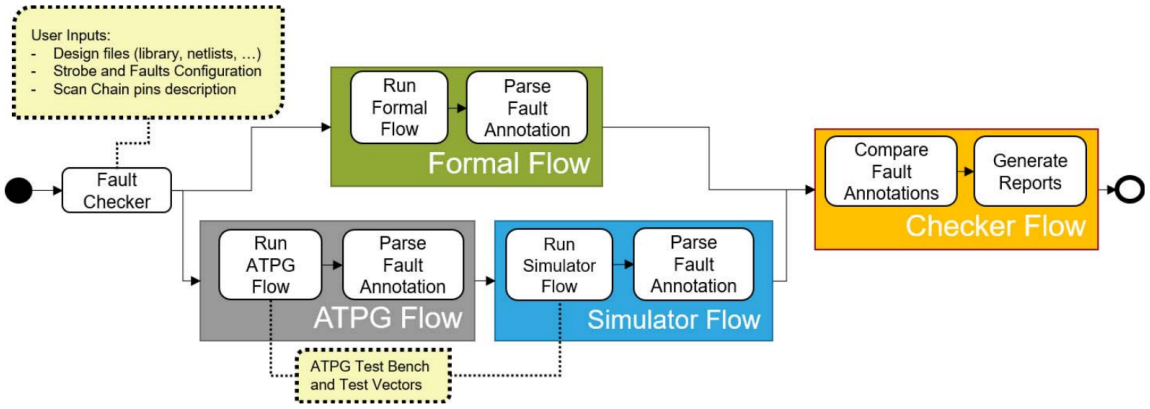


Figure 3.3: Combination of fault analysis technologies [10]

Another key aspect in fault simulation is the classification of undetected faults. The primary challenge with such faults lies in the uncertainty regarding whether they are safe or dangerous, necessitating either enhanced testing or expert analysis for further classification. In [35], [36], a strategy is introduced wherein code coverage obtained during functional verification is employed to classify faults as safe. The behavior of code coverage is then automatically translated into formal properties to constrain the environment. Without any modifications to the design, this methodology itself succeeded in enhancing the ISO 26262 metric sufficiently to elevate a CAN controller design from ASIL B to ASIL C. [37] also discusses about the usage of software-based self tests to elevate fault coverage levels in general purpose GPUs used in automotives. Test programs are refined based on initial fault coverages until they meet the desired coverage requirements, in addition to identifying Safe faults through formal methods.

The methodologies and techniques mentioned above provide interesting insights into improving fault injection simulations and classifications. However, there remains limited discussion on the classification of the results and the effectiveness of the tools in achieving their intended purposes. Additionally, there is a need to verify the accuracy of the tool results at various abstraction levels, especially considering that functional safety EDA tools now extend fault injection support to the RTL stage.

3.3. Safety Mechanisms

Safety mechanisms refer to supplementary logical components integrated into the design, contributing to the "safety" aspect of the functional safety process. While these components don't directly contribute to the primary functionality of the system, their role is crucial in identifying potential errors within the system and assessing its ability to detect them. This section explores prevalent safety mechanisms employed in automotive chips, shedding light on their current advancements.

3.3.1. Dual Core Lock Step (DCLS)

The primary idea of DCLS lies in redundancy wherein two processor cores are initialized with the same states and fed with identical inputs. The outputs are then compared to check for any discrepancies and recovery to safe state is initiated, if required. To remove common mode failures, temporal diversity is added by delaying the redundant core for a few cycles by inserting shift registers. Another way adopted to avoid common failures is to implement the redundant core in a different way, for example, by incorporating different ALUs or physical design implementations. Two instances of a comparator logic are also implemented to avoid faulty behaviour from a single one.

As shown in Figure 3.4, an ARM Cortex-M33 based dual core lockstep processor [38] consists of two identical processor cores (`TEAL`), along with two comparator logic (`tealdccm`), a DCLS controller module and resynchronization/delay flip-flops (`tealdcctl`) to provide the temporal diversity. The same clock is fed to both the cores, with the fault indicator signals being indicated from DCCMOUT[0] and DCCMOUT2[0], from the main comparator and redundant comparator respectively. THE DCCMINP signals are used to deassert the fault indicator signals when required. Such a DCLS logic helps the system detect logic failures and provides good reliability to the overall system.

In [39], the authors discuss about different memory architectures in DCLS systems. Independent memory architectures allocate separate memories to the two processors and thus can be optimized to run independently when there is no need for checking errors. Storage units are copied, meaning that memory errors could also be detected. Shared memory architectures have a single memory for both the processors in a master-slave system, with the master being able to access external modules as well. This configuration offers advantages such as reduced cost and area, while eliminating the need to resolve conflicts arising from shared components. The authors also discuss checkpoint-based fault tolerant designs [40] which use a combination of both hardware and software to establish checkpoints within the program. These checkpoints serve as reference points to rollback to in the event of an error occurrence. They propose a hardware-based DCLS using checkpoint-based designs, which provide advantages such as improved fault tolerance rates, performance and recovery time with little area overhead. In [41], virtual cores are used to create two lockstep processors using two 5-stage pipelines in an interleaved fashion to overcome common mode failures. Further, with the redundant architecture running on a different pipeline, the system can also detect single point failures.

DCLS processors offer commendable reliability, albeit at the cost of requiring a larger area footprint. However, as seen above, variations of the concept can be employed to optimize the system according to specific requirements, particularly in applications necessitating high levels of safety.

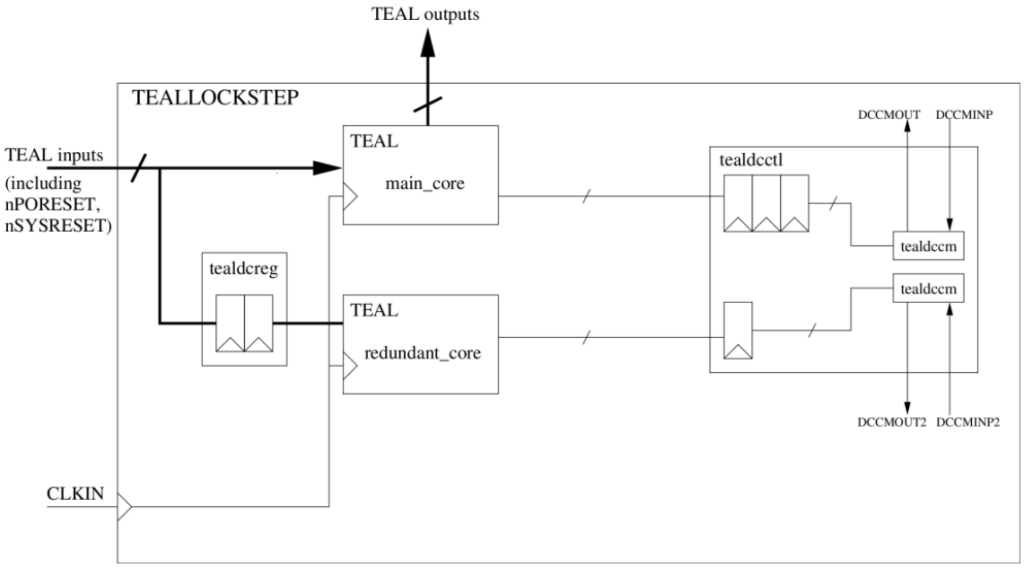


Figure 3.4: Cortex-M33-based DCLS processor [38]

3.3.2. Triple Modular Redundancy (TMR)

Triple modular redundancy (TMR), also known as triple-mode redundancy, is a fault-tolerant technique that falls under the category of N-modular redundancy. In TMR, three systems independently execute a process, and the output from each system is fed into a majority-voting system. This majority-voting system then produces a single output based on the results of the three systems. If one of the three systems fails, the remaining two systems can detect and correct the fault, ensuring uninterrupted operation.

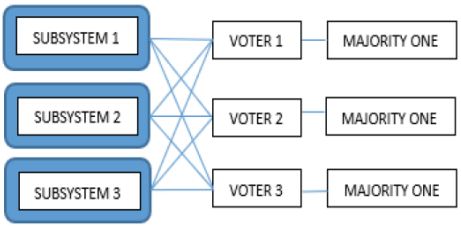


Figure 3.5: Triple voter mechanism in TMR [42]

To mitigate potential failures arising from voter circuits, a triple voter mechanism is commonly implemented, as discussed in [42], and shown in Figure 3.5. In [43], the authors discuss the reliability of a TMR system with a spare component. Such a configuration involves a spare processor accompanied by additional logic, including disagreement detectors, controllers, and switches. Illustrated in Figure 3.6, disagreement detectors, implemented with simple XOR gates, assess whether a processor's output aligns with the voter output. Switches, managed by the controller, dictate the processors participating in the voting process. Employing 2-out-of-3 majority voters, the study suggests that this system typically yields superior reliability outcomes compared to 3-out-of-5 or higher redundancy systems solely by integrating this additional logic.

In [44], the authors propose a novel concept called Triple Modular Temporal Redundancy (TMTR) system, which incorporates a checkpoint scheme to establish temporal redundancy. They develop a reliability model to assess the system's performance under both independent and correlated faults, determining the optimal number of checkpoints required to improve reliability. Unlike traditional TMR systems, which struggle to recover from faults in two or more systems simultaneously, the TMTR system can mitigate such failures by rolling back to the latest checkpoints, thereby enhancing overall reliability.

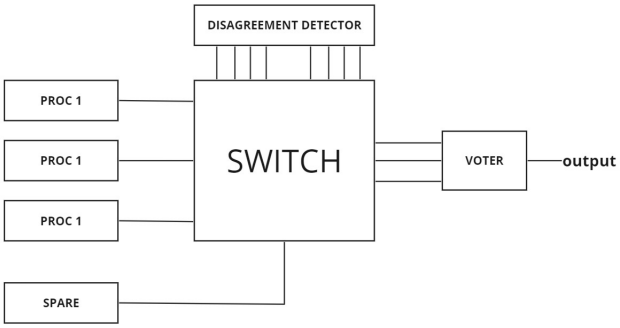


Figure 3.6: TMR system with spare

3.3.3. Built-in Self Test (BIST)

The primary motivatoin behind Built-in Self Tests (BIST) lies in its ability to provide in-field diagnostic functionalities for automotive Electronic Control Units (ECUs). BISTs are increasingly pervasive in automotive System-on-Chips (SoCs), particularly within safety-critical or high-reliability systems. Typically, BISTs come in two variants: Logic BIST (LBIST) and Memory BIST (MBIST). MBISTs are generally less cost-effective compared to other memory-related safety mechanisms like Error Correcting Codes (ECC), parity, and Cyclic Redundancy Check (CRC). Additionally, MBISTs may impose higher area and power overheads than their counterparts, further reducing their attractiveness for automotive applications. Therefore, in this section, we focus primarily on LBIST technologies and techniques commonly employed.

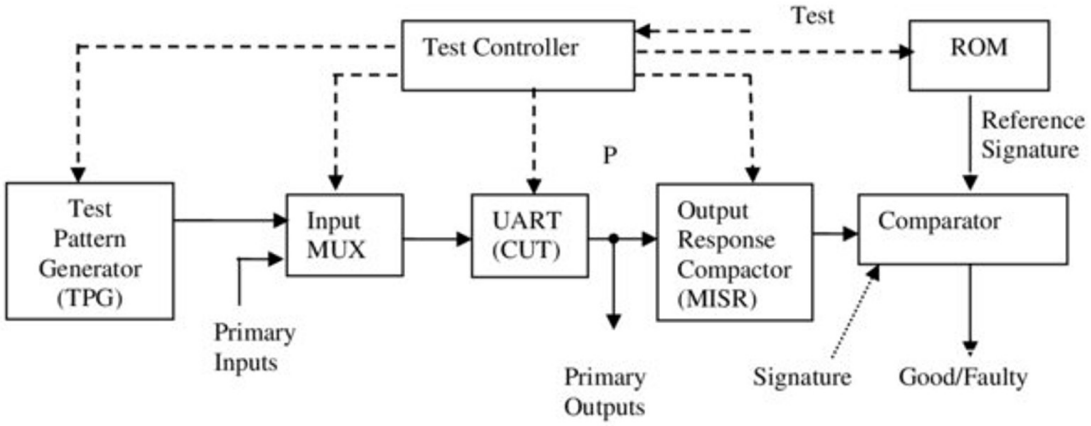


Figure 3.7: Architecture of LBIST

The basic architecture of LBIST is shown in Figure 3.7. It consists of a Test Pattern Generator (TPG) which is responsible for generating test patterns that are applied to the circuit under test (CUT). These patterns are designed to detect faults within the circuitry by exercising various paths and conditions. The test patterns along with primary inputs are fed to the CUT to produce the outputs, which are then compacted to a signature value using Multiple Input Signature Register (MISR). The Read-Only Memory stores the expected responses or signatures that are generated by the TPG. These expected responses are used by the Comparator to verify the correctness of the circuit's operation. The test controller coordinates the operation of the TPG, Comparator, and ROM, ensuring that test patterns are generated, applied to the circuit, and compared against expected results in a systematic manner. Additionally, the Test Controller manages the overall test flow and controls the timing and sequencing of test operations. One of the primary research areas lies in the optimisation of TPG to reduce the test time and memory, while at the same time increasing fault coverage.

A survey of different TPGs commonly used in LBISTs is presented in [45]. The survey categorizes

TPGs into four versions based on the presence of Test Point Insertion (TPI) or scan compression: 1) No TPI or scan compression is used, 2) Only scan compression is used, 3) Only TPI is used, 4) Both TPI and scan compression are used. Scan compression involves breaking down long scan chains into multiple shorter scan chains to reduce test times, albeit with a trade-off in fault coverage. TPI, on the other hand, involves inserting test mode Flip Flops (FFs), known as test points, into the circuit. These test points are categorized into control points (for forcing values into internal nodes) and observe points (for capturing values of internal nodes). Moreover, two separate LBIST strategies are applied on top of this: 1) "Full" strategy which uses an On-chip pattern generator that generates ATPG patterns to achieve the highest reachable test coverage, 2) "Top-off" strategy which applies a set of Linear Feedback Shift Register (LFSR)-generated patterns and runs ATPG for remaining faults. The results shown in Figure 3.8 indicate that Test Point Insertion (TPI) enhances fault coverage while potentially introducing delays in critical paths, and not meeting timing requirements. Conversely, Scan Compression reduces coverage but substantially reduces test duration.

	Basic		SC[1]		TPI		TPI + SC[1]	
Design area / μm^2		347,379		357,764		353,922		364,454
SC/TPI overhead		0.00%		2.99%		1.88%		4.92%
# Chains		32		384		32		384
# Scan cells		5,754		5,754		5,843		5,843
Test pattern type	Full	Top-off	Full	Top-off	Full	Top-off	Full	Top-off
Test coverage	97.05%	97.05%	96.38%	96.38%	97.06%	97.06%	96.52%	96.53%
LFSR coverage	-	93.27%	-	92.05%	-	96.59%	-	95.76%
# Test patterns	1,189	232	812	279	1,139	216	716	411
Test time / clock cycles	215,390	1,309,173	13,008	116,480	209,760	1,327,928	12,189	126,004
On-chip TPG area / μm^2	217,831	72,467	147,245	65,897	186,516	9,267	143,738	80,735
On-chip TPG overhead	62.71%	20.86%	41.16%	18.42%	52.70%	2.62%	39.44%	22.15%
Total overhead (with SC/TPI)	62.71%	20.86%	42.39%	18.97%	53.69%	2.67%	41.38%	23.24%

1. SC = scan compression

Figure 3.8: Test coverage, Test Time, and Area overhead for different DFT techniques and LBIST strategies [45]

In addition to this, there are various BIST methods which are widely used in different systems. Weighted Pseudo Random Test Pattern generators [46] are used to generate test patterns by assigning weights to bits or combinations of bits to enhance fault coverage and improve fault detection capabilities during testing. Bit Flipping BIST [47] employs a bit flipping function, utilizing the LFSR state to alter the value propagated into the scan path, thereby enhancing the randomness of test patterns. This technique effectively transforms random test patterns into a comprehensive test set, improving fault coverage in the testing process. BIST reseeding [48] involves encoding deterministic patterns into smaller vectors known as seeds, which are then loaded into the LFSR and expanded into desired test patterns. This process typically involves solving a linear system of equations to algebraically represent the linear expansion of the LFSR and generate the seeds required for the testing procedure. Further, to achieve higher fault coverages, Mixed mode BIST [49] combines the use of LFSR-based pseudo-random patterns with deterministic patterns. The mapping logic is a key technique in mixed mode testing, helping identify patterns within the pseudo-random sequence that do not detect new faults and map them to deterministic patterns for enhanced fault detection capabilities.

3.3.4. Software Test Library (STL)

A Software Test Library (STL) comprises software safety mechanisms that supplement hardware safety features to detect random hardware faults. STLs offer a means to conduct processor testing while mitigating the effects on the application. They can initiate tests during system startup to ensure proper functionality before the safety application commences execution. Alternatively, they can be set to test during application execution without necessitating processor rebooting afterward. These tests can be activated either as a comprehensive block or in brief bursts, enabling checks to be conducted whenever time allows, thereby minimizing application disruption.

A key benefit of STL is that it does not incur additional hardware overhead. Users are not required to have a complete understanding of the STL tests; they can simply test the products with the given STL to assess fault coverage. However, the generation of STLs is primarily a manual process, and need to

conform to several constraints, as also highlighted in [50]. STLs intended for in-field testing must comply with the operating system specifications without interrupting the regular operation of the device's application. This integration of STLs with simple and cost-effective hardware safety mechanisms has demonstrated increased fault coverage with less area overhead, as observed in an SoC design in [51].

3.3.5. Error Correction Codes (ECC)

Error correction codes (ECC) are methods employed in functional safety to detect and correct errors that may occur during data transmission or storage. These codes involve adding extra bits to the transmitted data to create redundancy, enabling the receiver to detect and, in some cases, correct errors that may have occurred during transmission. In broad terms, ECC falls into two categories: Block codes and convolutional codes. Block codes involve breaking messages into fixed-sized bit blocks, to which extra redundant bits are added for error detection and correction. Convolutional codes, on the other hand, operate on variable-length data streams, where parity symbols are derived through the application of a Boolean function as a sliding window moves across the data stream. There are different types of ECC algorithms, including Hamming codes, Reed-Solomon codes, and BCH (Bose-Chaudhuri-Hocquenghem) codes, each with its own method of encoding and decoding data to detect and correct errors.

Hamming codes are one of the most popular ECCs, which can achieve Single Error Correction and Double Error Detection (**SEC-DED**). The first step in Hamming code encoding is to calculate the number of redundant bits, r in a data of m bits, and follows the equation:

$$2^r \geq m + r + 1 \tag{3.1}$$

Considering a Hamming code for 8-bit data, 4 redundant bits are enough to encode the information, as per the above equation (2^4>8 + 4 + 1). The redundant bits are placed at indexes which are powers of 2 (1,2,4,8 etc.) These bits, denoted as r_i, are computed using XOR operations across different bit positions. Each r_i represents the even parity based on its bit position, covering all positions with a binary representation that includes a 1 in the i^{th} position, excluding the position of r_i. For instance:

- r_1 is the parity bit for all data bits in positions where the binary representation includes a 1 in the least significant position, excluding 1 (3, 5, 7, 9, 11, and so on).
- r_2 is the parity bit for all data bits in positions where the binary representation includes a 1 in the position 2 from the right, excluding 2 (3, 6, 7, 10, 11, and so on).

If we have 8 data bits, each represented by d_i, then the final encoded data bits (e_i) will be as shown in Figure 3.9, and calculated according to Equation 3.2. Now in order to decode the same, we take the encoded data and apply the same XOR operations to get back the bits, dec1, dec2, dec4 and dec8 (ECC bits). These bits combined form a 4-bit number {dec8, dec4, dec2, dec1}, which is then XOR-ed with the original encoded data bits at these positions ({e8, e4, e2, e1}) to retrieve the index where the error has occurred. This bit is then flipped to give the correct decoded data.

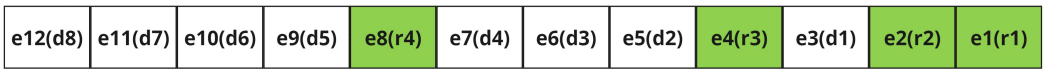


Figure 3.9: ECC encoding for 8 data bits using Hamming code

$$\begin{aligned} e1 &= e3 \oplus e5 \oplus e7 \oplus e9 \oplus e11 = d1 \oplus d2 \oplus d4 \oplus d5 \oplus d7 \\ e2 &= e3 \oplus e6 \oplus e7 \oplus e10 \oplus e11 = d1 \oplus d3 \oplus d4 \oplus d6 \oplus d7 \\ e4 &= e5 \oplus e6 \oplus e7 \oplus e12 = d2 \oplus d3 \oplus d4 \oplus d8 \\ e8 &= e9 \oplus e10 \oplus e11 \oplus e12 = d5 \oplus d6 \oplus d7 \oplus d8 \end{aligned} \tag{3.2}$$

Hamming codes offer a straightforward yet efficient approach to error detection in memory locations, presenting a cost-effective solution. Building upon this fundamental redundancy concept, numerous

ECC algorithms have been devised to enhance reliability in automotive applications. An example of the aforementioned ECC approach is also seen later in the report during the implementation of a FIFO protected by the ECC safety mechanism. An ECC application is also present in the final IP to be tested using the verification methodology to be developed.

3.3.6. Error Detection Codes (EDC)

Error detection codes (EDCs) are mechanisms employed to detect errors that may occur during data transmission or storage. They are crucial in ensuring data integrity and reliability in various systems. There are different EDCs commonly used, which are discussed briefly below:

- **Parity**: Parity is one of the simplest forms of error detection. For example, in even parity, an extra bit (the parity bit) is added to each unit of data (e.g., byte or word) such that the total number of ones in the data, including the parity bit, is always even. If a single bit error occurs during transmission or storage, the parity check detects it by comparing the number of ones in the received data with the expected parity.
- **Checksum**: A checksum is a value calculated from a block of data using a mathematical algorithm. For instance, in Internet Protocol (IP), the Internet Checksum is computed for each packet to ensure data integrity during transmission over networks. The receiver recalculates the checksum from the received data and compares it with the transmitted checksum. If they do not match, an error is detected.

  Cyclic Redundancy Check (CRC) is a robust error detection technique widely used in digital communication systems, and can be considered as an algorithm for calculating checksums. For example, Ethernet and Wi-Fi protocols employ CRC to detect errors in transmitted data frames. A polynomial algorithm generates a CRC checksum based on the data being transmitted. The receiver performs the same CRC calculation and compares the computed checksum with the received CRC checksum. If they differ, an error is detected.

  Various other algorithms for computing checksums, such as MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), Internet checksum (RFC 1071), etc. are widely used in different solutions as well.

In summary, error detection codes such as parity, checksum, and CRC play a crucial role in identifying errors during data transmission or storage, ensuring the integrity and reliability of digital systems. They are fundamental in various applications, including networking, storage systems, and digital communication protocols.

3.4. Research Gap and Conclusions

In light of the research question at hand, we looked at different methodologies using FuSa EDA tools and alternative approaches as well. Testbench based fault injection is not very scalable to larger complex designs and requires a lot of manual work. Emulation based fault injection is typically carried out using FPGAs, and hence not in line with our thesis goals. Further, fault injection simulation is the preferred methodology suggested by ISO 26262 for FuSa. So, it becomes evident that fault simulation EDA tools, with their variety of capabilities and features, become the go-to methodologies for FuSa verification solutions.

While a lot of the approaches focus on extracting Safe faults initially with the help of formal tools and ATPG in order to save time on simulating these faults, it is yet to be seen whether safe faults can be extracted from RTL designs as well. Further, we see discrepancies in the classification of Safe faults from these tools. This begs the question - **will two FuSa EDA tools give the same classification results when running fault injection simulation on different points of an RTL design?** There is no evident information on the accuracy of fault simulation results of different tools, or a comparison between different features and optimisations of different tools. Can the preference for one tool over another help speed up fault campaigns and yield accurate results?

Second, once we get the results, how do we improve the diagnostic coverage - by improving tests or by developing additional safety mechanisms? Do the tools provide the necessary information in order to make an informed decision? In the next few chapters, we aim to answer these research gaps. We will begin by presenting a detailed comparison of FuSa fault simulation

EDA tools. Subsequent chapters will delve into other areas of research gaps, exploring avenues for improvement.

4

Comparison of existing EDA tools

The three primary tools used for fault simulation purposes are Xcelium Fault Simulator (XFS) (Cadence), VC Z01X (Synopsys) and Austemper/Kaleidescope (Siemens/Mentor Graphics), and is widely adopted in many solutions for functional safety verification. As introduced before, there is a lack of research on why we should favour one tool over another. Keeping that in mind, this thesis looks at a quantitative and qualitative comparison of these tools, by applying the appropriate setup on reference designs and analyzing the corresponding results. Since a license was not available for Austemper and Kaleidescope during the course of the thesis, it was not possible to include this tool in the comparison analysis. In the following sections, we look at the general flows for XFS and VC Z01X. We automate the verification flow using these two tools, use them on reference designs to analyze the results and make comparisons.

4.1. Overview of tool flows

A top level flow of general verification methodologies for XFS and VC Z01X are outlined in Figure 4.1, and follows a generic approach as outlined in Section 2.3. The tools offer different features with regards to ISO 26262 compliant FuSa verification and utilities for optimisations and ease of usage. In this section, we introduce the individual flows of the tools, along with their features to further compare them in the subsequent sections.

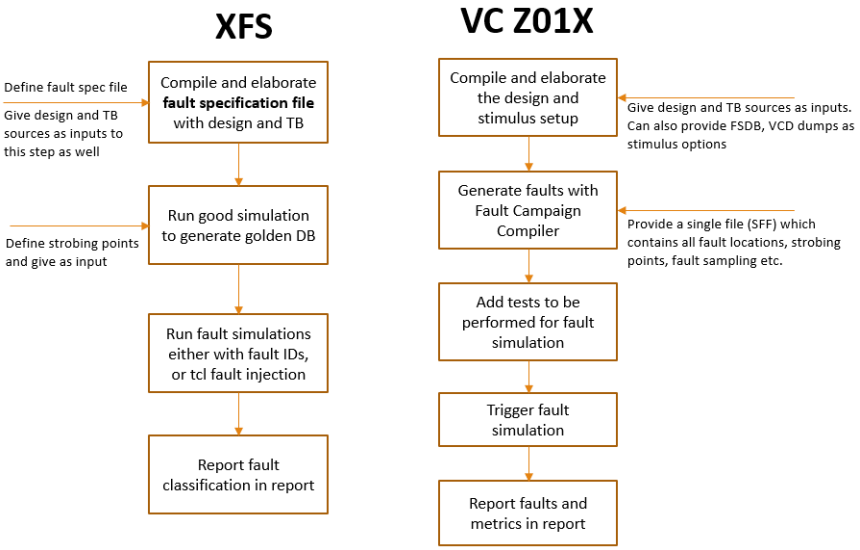


Figure 4.1: Overview of XFS and VC Z01X flows

4.1.1. Xcelium Fault Simulator (XFS) flow

The Xcelium Fault Simulator (XFS) is a Functional Safety EDA tool which facilitates fault injection simulation with the reuse of existing functional verification environments. XFS supports SA0 and SA1 faults, which can be applied to any kind of signal in the design. In transient faults, XFS supports Single Event Upsets (SEU) which can be applied only on the outputs of sequential elements - memories, flip-flops and latches. It also supports Single Event Transients (SETs) which can be applied on any signal by inverting its current value and holding it for a required period of time. Therefore, according to ISO 26262 specification, it covers the necessary fault models required for FuSa verification. As outlined in Figure 4.1, the different steps involved in the XFS flow are detailed below:

1. **Definition of fault specification file**: The first step is to define the targets where we want to instrument or inject faults. In order to do this, a fault specification file needs to be created which defines all possible locations of fault instrumentation. The scope of fault targets could be any module name, signal name or hierarchical path to an instance, which can be either specified using the testbench or module hierarchy. If a module name or an instance is provided, by default, all ports, nets, variables and registers inside them are identified as faultable nodes. It is also possible to include all possible faultable nodes from all hierarchies under a given module using three dots as a suffix after the fault target name.

   It is also possible to exclude certain targets in the fault specification file. For example, if there are certain blackbox modules where we do not want to instrument faults, we can include them as exclusion targets. Particular signals could also be excluded from instrumentation in this regard. In this stage, XFS provides several switches in order to identify the type of faults to be instrumented:

   - **Net faults** can be switched on to allow for fault instrumentation on wires. For example, if we instrument a port fault (M.A in Figure 4.2), the fault by default will not be instrumented on the wire (M.N) connected to that port. However, if we use the net switch, we will be able to instrument the fault on the wires connected to the port as well.

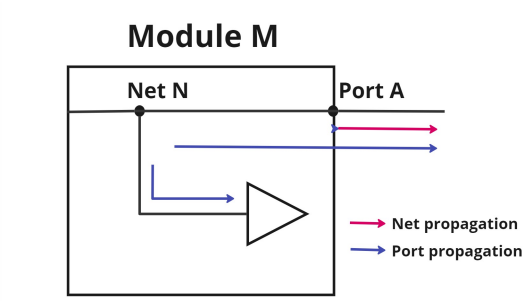


Figure 4.2: Net faults with XFS

   - **Port faults** can be switched on in order to instrument faults at ports of modules or cells only. Internal nets and regs will not be considered as fault targets in this case.
   - **Ports of cell instances** can also be configured as targets, in which case non-cell module ports and nets/regs declared inside a module or cell will not be considered for fault instrumentation.
   - **Sequential elements** only can be identified as faultable nodes, examples including variables driven by an `always` block at RTL or a sequential User Defined Primitives (UDP) at gate level.
   - There is also a switch to specify the type of faults we want to consider - SA0, SA1, SET, SEU or all. Depending on the fault type, different points of the fault target will be considered for instrumentation. For example, if we mention the fault type as SEU, only sequential element outputs will be instrumented.

2. **Elaboration of fault specification file**: The fault specification file needs to be elaborated along with the necessary design files and verification testbench. This will create a fault database containing all the nodes where faults are to be injected along with the fault type. There are several

options which can be passed along with this elaboration command. For example, gate collapsing capabilities and optimisations can be turned off, or untestability/safe fault identification can be turned off. Such switches can allow the user to run faults on all possible nodes of the design when required.

3. **Fault strobing**: Before we continue with running the good simulation, signals in the design need to be identified for observing functional and safety mechanism outputs. Such points, also known as strobing points, are further divided into two categories - functional and checker strobes. It is also however possible to define all strobe points without any distinction. A strobe point can be **Detected**, meaning it has a value mismatch between the good simulation and faulty simulation run. It could be **Potentially Detected** - meaning that there is a signal value change for the specified strobe point, but with an unconfirmed value (x,z,u,h,l). Finally, it could be **Undetected**, meaning it has no difference in value between the two runs.

   The fault strobes are placed in a tcl (Tool Command Language) file, which is then later used in the fault injection campaign for both the good and fault simulation runs. In this tcl file, it is either possible to mention a single strobe list without any distinction of functional and checker strobes, or to list out both types of strobes. The latter is recommended in order to classify faults according to ISO 26262 classification.

   Fault strobing also provides certain switches, such as mapping potentially detected values or stopping simulations. The mapping switch offers two options: **pessimistic** and **optimistic**. In the pessimistic mode, potentially detected results for functional strobes will be marked as detected, while checker strobes will be marked as not detected. Conversely, in the optimistic mode, the opposite scenario occurs. There is also a `stop_on` switch which tells when the simulation is to be stopped. Simulation could be halted if a fault is detected at any strobe point, or if it is potentially detected, or the simulation could be even allowed to continue. This provides options to configure the simulation according to the requirements.

   There is also the option of virtual strobes, which are system tasks that can be put anywhere in the design or testbench to look for complex patterns and configure them as strobes. All the strobing information can also be captured, and then later reported to see the changes in signal values of the defined strobes.

   In order to comply with **Fault Tolerant Time Interval**, the fault strobing option allows a checker delay interval switch, specifying the duration for monitoring checker outputs following the detection of an injected fault at one of the functional outputs. The simulator stops monitoring the checker outputs upon reaching the specified time, regardless of whether a fault is detected.

4. **Serial Fault Simulation**: The next step in the flow is to run good simulation in order to generate a golden database to compare the fault simulation runs. XFS provides support for two engines: serial and concurrent. The serial engine initiates the good simulation first, followed by the triggering of fault simulation runs. In contrast, the concurrent engine executes both the good and faulty simulations simultaneously.

   The good simulation is triggered along with the strobing tcl file to create a reference golden database. Necessary arguments and tests need to be provided with the command. All the identified fault nodes along with their fault IDs are generated after the good simulation. At this point, we can now run fault simulations by selecting a fault ID or any random ID from the generated fault list and simulate the fault with the given tests. With this method (also known as **random fault campaign**), for every fault ID, a separate command needs to be triggered to run fault simulation.

   However, there is another method (**targeted fault campaign**) by which we can perform fault injection through a tcl file. This file contains fault injection commands for each fault node, specifying the fault type, injection time, and a stop severity similar to the `stop_on` switch. Additionally, an option exists to specify a start and end time frame, during which the simulator randomly selects a time unit for fault injection. This approach consolidates all fault injection commands into a single file, allowing it to be executed once with the simulation command. One thing to be noted is that after every command, a `run;reset;` needs to be appended, in order to run the simulation and then reset it once completed. These commands can be written manually, however, Xcelium also provides the Xcelium Fault Set Generator (xfsg) utility in order to automate the generation

of fault injection commands in the tcl file. The fault simulation proceeds by providing the tcl file as input and simulating the faults sequentially, one at a time.

5. **Concurrent Fault Simulation**: Instead of running the good and faulty simulations separately, the concurrent engine runs them in parallel, with the capability of injecting and simulating multiple faults simultaneously in a single fault run. This leads to increased throughput as compared to serial fault simulation. The concurrent fault simulation switch needs to be provided along with the strobing and fault injection tcl files in a single invocation of the simulation command. The concurrent fault engine, however, supports only SA0, SA1 and SEU fault types. If SET faults are specified, they will not be simulated. Further, only zero delay gate-level designs are supported with the concurrent engine by default. In order to enable concurrent fault simulation at RTL, additional switches need to be provided. However, there are limitations with different Verilog RTL constructs as well. Also, SystemVerilog and VHDL is not supported with concurrent simulations.

6. **Fault reporting**: Xcelium Fault Reporting (xfr) is a post processing utility provided by XFS in order to report the results of fault injection simulation. The database containing the results of fault simulations are provided as an input to this utility, which also has several switches for different types of optimisations. The final report generated typically has the fault classifications for the given nodes, along with their injection time, observation and detection times. In case both functional and checker strobes are present, the classifications will be as shown in Table 2.5, with differing names. If a single strobe list is present, the classifications will be Detected, Potentially Detected and Undetected. Finally, a summary of the classification numbers is generated in a log file.

   XFS also assigns fault classification priorities, with Observed Diagnosed classification holding the highest priority and Undetected classifications the lowest. When merging SA results of faults from various directories, xfr evaluates each node and its corresponding fault classifications from different directories, annotating them with the highest priority. Regarding transient faults, xfr merges only those of the same type, injection time, and hold time.

As seen from the steps above, there are a few ways in which FuSa verification can be utilised with XFS, particularly with the serial and concurrent fault engines, and targeted/random fault campaigns. We will see the results of these techniques in subsequent sections.

4.1.2. Verification Continuum (VC) Z01X flow

Verification Continuum (VC) Z01X is the fault simulation EDA tool provided by Synopsys, which is developed on top of VCS (used for logic fault simulation), and also reuses functional verification testbench setup for fault simulations. VC Z01X, by default, uses a concurrent engine, meaning that it runs the good simulation with large number of faulty simulations in parallel. However, it also provides support for a serial engine, if and when required.

As depicted in Figure 4.1, the details of the VC Z01X flow and the corresponding features are elaborated below:

1. **Compilation and Simulation of design**: In the first step, the VC Z01X compiler is called by the vcs command in order to create the simulator executable and the design database. In this step, all necessary design and testbench files are to be provided. If the provided files are of different types (Verilog, VHDL, SystemVerilog), this step is usually broken down into two smaller steps - one for analysing the different types of files separately, the second for elaborating the same with vcs command. Necessary switches for different constructs in different versions of HDL used also need to be provided.

   In addition to testbench support, VC Z01X also supports external stimulus options such as Fast Signal Value Dump (FSDB), Value Change Dump (VCD), eVCD and forcelist. In this case, the stimulus options must be provided along with the elaboration command in order to prepare for external stimulus. Also, in order to prepare for fault simulation, the `fsim` switch needs to be added. This switch provides options for enabling different types of faults, such as port, primitive or array faults, among others. This is a required switch during elaboration, in order to enable fault simulation.

During this stage, a good simulation is not necessary but can be run using the simulator executable generated after elaboration. Essential test arguments must be provided along with the simulation command.

2. **Generation of faults with Fault Campaign Compiler**: In the next step, we provide a **Standard Fault Format (SFF)** file to the **Fault Campaign Compiler (FCC)** of VC Z01X in order to generate the fault database. The SFF file is the crux of fault simulation in VC Z01X, which consists of all the functionalities and features. In this part, we discuss the different sections of SFF file, and their usage.

   VC Z01X has an extended default classification list for faults, providing for better debugging. It provides multiple fault groups, which in turn has multiple fault classes. For example, there is a **Detected Fault Group**, which consists of three different fault classes. The first class corresponds to faults whose good and faulty simulation values do not match. The second class is for faults which causes Verilog assertions to fire, resulting in a call to `$fatal` or `$error`. The third class refers to faults where `$stop` or `$finish` is encountered. Various fault groups within the tool offer additional features regarding fault classification rationale. The tool itself facilitates the identification of Untestable faults or Safe faults, accompanied by explanations categorized under different classes. Additionally, fault classes may indicate whether a fault remains unpropagated due to static workload values or is blocked by another signal. These classifications improve understanding of fault classification reasons, enabling improvement of tests or design modifications.

   The different sections of the SFF file are discussed below:

   - The **Test Information** section contains details about the individual test runs along with the fault counts obtained from each test.
   - VC Z01X allows for **User Defined Fault Statuses (UDFS)**. According to ISO 26262, functional and checker outputs should both be monitored to get the classifications of both classes. However, the default classes of VC Z01X do not provide such a categorization. Hence, custom classes can be defined in order to accommodate the same. Further, existing fault classes can also be redefined to signify other definitions.

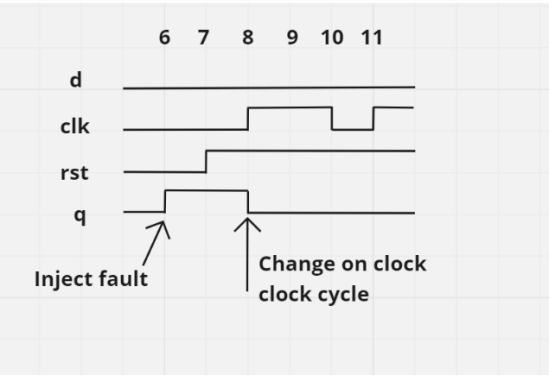
     It is important to know how different fault statuses interact with each other. This is defined with the help of a Promotion Table, which is a 2D array of the interaction between the fault status of a previous simulation compared to that of a new simulation. There is a default Promotion Table defined by VC Z01X, consisting of the interaction between defined classes. If we are using UDFS, we need to define our own Promotion Table in the SFF file in tabular form.
   - The **Coverage section** enables the definition of equations for fault coverage or ASIL metrics in the final report, using the fault classes previously defined. Following the completion of all fault simulation runs and the availability of fault classification numbers, the coverage equations specified in the SFF file will be automatically calculated and included in the final report.
   - The **FailureMode section** within the SFF is utilized to link fault observation and detection data with each specific failure mode, facilitating FMEDA calculations. It contains a list of observation and detection points/signals in two subsections, which can be strobed to find out differences in good and faulty simulation values. With this FailureMode section, two system functions - `$fs_observe` and `$fs_detect` can be used to find out the status of the defined signals - whether they have been detected, potentially detected or undetected.

     In addition, a **SafetyMechanism section** can also be defined and then included inside a FailureMode instead of individually specifying the detection signals. This allows for reusability of SafetyMechanism in different Failure Modes.
   - The **FaultGenerate section** defines the list of fault targets to be instrumented, along with the type and location of fault. VC Z01X allows for faults to be placed at **array (ARRY), port (PORT), prim (PRIM), variable (VARI), wire (WIRE) and flop (FLOP)**. All these locations are supported for SA0 and SA1 faults. However, for transient faults, only **ARRY, PORT, VARI, and FLOP** classes are supported. There is also a location filter list which could be specified in order to place the faults at INPUT, OUTPUT, INOUT, REG,

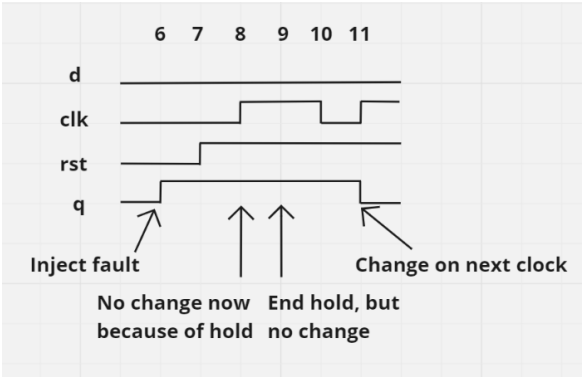
LOGIC, etc. This allows for more customisation with respect to the type of faults we want to instrument. With the help of wildcard syntax, we can also include fault targets of multiple hierarchies under a particular module or instance. The FCC interprets the FaultGenerate section and figures out the different fault locations based on the provided options.

There are two types of transient faults supported by VC Z01X - transient toggle and transient hold. Transient toggle injects a fault by flipping a bit at the specified cycle time, thereby inverting the current value at the specified location. The fault location will be updated again according to the input at the next clock cycle. As shown in Figure 4.3a, a transient toggle is injected on q at 6^{th} cycle. The value is inverted and changes back to 0 at the positive edge of the next clock at 8^{th} cycle (based on input d = 0).

Transient hold inverts the value of fault location and holds it until the specified cycle. The value will be updated at the next clock cycle after the hold has been released. As shown in Figure 4.3b, the transient hold is injected on q from 6^{th} to 9^{th} cycle. Therefore, the inverted value is sustained until the next rising edge of the clock, when it reverts back to 0, as driven by d being set to 0.



(a) Transient toggle at cycle 6



(b) Transient hold from 6 - 9 cycles

Figure 4.3: Transient fault behaviour in VC Z01X

Transient faults introduce the concept of cycles. The specifics of each cycle must be outlined within the FaultGenerate section of the SFF file. This includes defining the total duration of the cycle and an offset for fault injection. It's essential to understand that this cycle does not necessarily align with the clock of the design or testbench but can be configured to match it.

VC Z01X supports a very useful feature of **Sampling** in the FaultGenerate section. Fault sampling is a method used to reduce the number of simulated faults while still approximating the coverage outcome achievable through simulating all faults. There are three approaches for determining the sample size. **Percentage-based sampling** calculates the fault count based on the user-provided percentage. **Fixed number sampling** allows users to directly define the desired sample size. **Confidence Interval sampling** employs mathematical calculations to ascertain the necessary fault count, considering a margin of error (confidence interval) and user-specified confidence level.

The required sample size for confidence based sampling is calculated using statistical formulas as described below:

$$\text{New sample size} = \frac{\text{sample size}}{1 + \frac{\text{sample size - 1}}{\text{population}}} \tag{4.1}$$

where

$$\text{sample size} = \frac{z^2 * \text{p} * (1 - \text{p})}{C^2}$$

population = total number of faults

p = coverage estimate expressed as a decimal. Default value is 0.5, if coverage estimate is not provided

z = standard Z score based on confidence level

C = confidence level

Given a design which has 100,000 faults, a confidence interval of 1% and a confidence level of 99% would require only 14,267 faults to adequately represent the design, as shown below in the calculations. Put differently, approximately 14% of randomly selected faults can reliably indicate, with 99% confidence, that the measured fault detection percentage falls within the interval of the true value plus or minus 1%. For instance, if the actual fault coverage percentage is 95%, the fault coverage will typically fall within the interval of (94% to 96%) 99% of the time.

$$\text{sample size} = \frac{2.58^2 * 0.5 * 0.5}{0.01^2} = 16{,}641$$

$$\text{New sample size} = \frac{16{,}641}{1 + \frac{16{,}641 - 1}{100{,}000}} = 14267$$

$$\text{population} = 100{,}000$$

$$C = 0.01$$

$$Z = 2.58(\text{for confidence level of } 99\%)$$

Fault sampling proves highly beneficial for efficiently reducing the fault space without compromising the fault coverage range for any given ASIL. This capability significantly reduces simulation time and accelerates fault campaigns.

This concludes an overview of the various features within an SFF file, which is used in a fault campaign to generate the necessary faults while conforming to specified optimizations and constraints. The FCC accepts one or more SFF files as inputs and generates the fault list for utilization in the fault campaign. Users have the option to define multiple fault campaigns using various SFF files to differentiate between campaigns.

3. **Addition of tests for fault simulation**: The next step is to define tests which will be used for fault simulation. Each test needs to be defined with the necessary arguments along with the simulator executable generated after elaboration. However, if FSDB files or any external stimulus is given, the simulator executable is not needed.

   Before starting fault simulation, various relevant parameters can be configured. These include selecting statuses to be run for fault simulation, disabling assertion checking and illegal file access faults, and enabling the fault dictionary, which captures the list of strobes detected during simulations.

4. **Fault simulation**: With the defined tests, we initiate the fault simulation with a simple `fsim` command. Upon triggering `fsim`, Fault Campaign Manager (FCM) executes toggle simulation (good logic simulation) and performs **testability analysis**. During toggle simulation, Z01X generates an FSDB file to capture the data utilized during testability analysis. Leveraging the information acquired from toggle simulation, FCM assesses the optimal test for detecting the maximum number of faults during testability analysis. Subsequently, the highest ranked test is executed, and this iterative process continues until all tests have been exhausted.

5. **Fault reporting**: Results can be reported based on a chosen Failure Mode (FM) from the SFF file or a specific fault campaign. The report is formatted as an SFF file and includes sections detailing the classifications of each signal and the final count of different fault classes and groups. At the end of the report, coverage equations are also calculated. VC Z01X further offers a useful feature enabling the documentation of reasons for specific Safe faults. For instance, if certain faults are deemed uncontrollable or blocked due to workload constraints, relevant information can be included in the report. Tests can then be changed to toggle the specified locations or design constraints can be evaluated. This facilitates improved debugging instead of categorizing faults as

Not Observed and Not Diagnosed, which would require manual inspection of waveforms to figure out the reason.

As evident from the process, VC Z01X provides a wide array of features and utilities to assist in FuSa verification. Subsequently, we will delve into comparing the two tool flows mentioned and analyzing the results obtained from using them on reference designs.

4.2. Comparison Metrics

To gain a better understanding of the tools, it is essential to execute the necessary flows on various designs and analyze the outcomes. However, prior to this, we must establish specific metrics for comparing the two tools. This approach will enhance our comprehension of the tool's suitability for FuSa verification. We establish the following metrics for comparing the two tools:

1. **Quality of results**: The first question that arises when employing EDA tools for fault classification is whether the tools are fulfilling their intended purpose. Evaluating the results of fault classifications and assessing their accuracy is paramount. This process becomes more challenging with larger, complex designs. Therefore, it's important to execute the flow on smaller designs for manual result analysis to ensure that fault classifications and coverage figures are indeed correct. Additionally, it's essential to verify if different types of faults are being simulated and if the entire fault space is covered. These factors collectively influence the quality of results.

2. **Comparison of individual features**: All the individual features of the two tools need to be taken into account while comparing. For instance, one feature of a tool may provide more utilities or optimisations as compared to the other, and therefore could be better utilized in the required flow. It could also be that certain features are supported by one tool, and not by the other. Thus, it is essential to carry out a comparative analysis of the features of the two tools to figure out the similarities and differences.

3. **Features for performance optimisation**: It's crucial to identify specific features in the tools that contribute to optimizing fault campaigns and expediting fault simulations. FuSa verification, especially for large designs with extensive fault space, is often time-sensitive. Any optimizations that enhance the performance of fault simulation can prove highly advantageous. For instance, exploring how concurrent engines enhance the speed of fault simulation in comparison to serial engines would be interesting.

4. **Feasibility of tool**: It becomes important to realize whether the tool supports all the necessary features required for fault simulation at a given abstraction level for any design. Thus, a combination of the factors mentioned above need to be taken into consideration while evaluating the feasibility of the tool to be used in FuSa verification flow.

5. **Run-time comparison**: Measuring the combined runtime of good and faulty simulations across different supported flows by both tools is essential to understand the time required for fault campaigns. For fair comparison, it is important to ensure an equal total number of faults when conducting fault campaigns with both tools. Furthermore, evaluating the runtime of the tools with feature optimizations also needs to be seen.

6. **Ease of usage**: The tool flow ought to be modular and adaptable across various designs with minimal adjustments, facilitating ease of usage for the user. When considering this metric, it is important to account for the required setup and the different modifications necessary to automate the flows for both tools.

4.3. Reference Design and Setup

To evaluate the outcomes of the two tools, we consider a few reference designs. Initially, we examine a basic full adder circuit, which is described using Verilog primitives to implement a structural design. Secondly, we utilize a behavioral RTL design of a 4-bit up counter. These designs serve as basic combinational and sequential circuits, offering an initial understanding of fault injection simulation with small-scale designs. The low complexity of the circuits allows for manual inspection of different fault classifications, if required.

Further, we analyze a First In First Out (FIFO) queue design equipped with various safety mechanisms. While the adder and counter circuits are simple designs, they do not contain any safety mechanisms and hence are not representative of functional safety designs. The FIFO design presents a more relevant representation with the memory array being protected by a Safety Mechanism (ECC) and logic being duplicated to detect additional faults. This facilitates an understanding of how safety mechanisms influence faults injected into the functional design, thereby mitigating the impact of hazardous faults. The subsequent sections delve into the design, verification, and FuSa setup of these circuits.

4.3.1. FuSa setup

A unified script is created to run three versions of the **XFS flow**, as outlined in Section 4.1.1. The fault target list, strobing file, as well as design and verification files remain common across the three flows for a particular design. However, if we want to use the same script for different designs, modifications are required in the fault target list, strobing files, design and TB sources passed to the script to accommodate the changes. The script offers different configurations, mentioned below:

- **Serial simulation with random fault ID**: This first configuration allows the user to run XFS flow serially by injecting the faults based on a random fault ID from the elaborated fault list. All the faults generated have a random fault ID, starting from 1 to the number of faults instrumented. Every fault injection is triggered with a single `xrun` command, along with the fault ID, injection time, fault type and the output directory. All different types of faults are run, and the output directories are merged with the help of `xfr` to generate the final report. **In this flow, a separate `xrun` command is required for every single fault.**
- **Serial simulation with `xfsg`**: In the second configuration, faults are injected sequentially, albeit without specific fault IDs. We use the `xfsg` utility in order to automatically generate fault injection commands, by giving in inputs for randomized fault injection times and fault types. An important thing to note is that xfsg utility does not allow for hold time as an input while generating SET faults. Hence, once the initial fault list is generated, we have to append a hold time for the required SET faults. A simple search and replace is done to inject SET faults with a fixed hold time. Finally, a run and reset is appended at the end of every fault injection command. **A single `xrun` command is then invoked with this fault injection list in order to simulate the faults.** Every fault is injected and run, after which the simulation is reset and the next fault is run again. This process is repeated until all faults are exhausted. All the results are eventually combined to generate the final report.
- **Concurrent simulation**: This flow also utilizes the xfsg generated fault injection list as the previous flow. However, a good and faulty simulation are not run sequentially. Instead, after the completion of elaboration phase, the good and faulty simulations are run in parallel with the help of the concurrent switch. The fault list generated with xfsg does not have to be appended with run and reset after every command, since the faults are run in parallel. Also, concurrent simulation does not support SET faults. Thus, in order to cover the entire fault space, we run serial fault simulation on the SET faults with xfsg flow. Both these flows are combined into a single script for a mixed concurrent-serial engine approach.

A unified shell script takes care of the aforementioned flows, with the help of arguments passed to the script (`xfs_flow.sh`). The different arguments supported are:

1. `serial`: run serial fault simulation
2. `concurrent`: run concurrent fault simulation (serial for SET faults)
3. `xfsg`: use xfsg generated fault list
4. `random`: use random fault ID from elaborated list for fault injection
5. `custom`: use a custom fault list for injection (do not use xfsg or random flow)
6. `sa`: simulate SA faults
7. `trans`: simulate transient faults

The script should be invoked with either `serial` or `concurrent` as an argument. Additionally, one out of `xfsg`, `random`, or `custom` must be specified as an argument. Both `sa` and `trans` can be passed as arguments to the script together. In order to run the first configuration, the command to run all faults would be : `./xfs_flow.sh serial random sa trans`. For the second flow, `./xfs_flow.sh serial xfsg sa trans` would be used. Finally, for the third flow, `./xfs_flow.sh concurrent xfsg sa trans` is to be run to generate results for all faults.

Based on the arguments provided, the corresponding fault simulation will be run. For example, if we provide a command such as:

```
./xfs_flow.sh serial xfsg sa
```

, a serial fault simulation will be run with the help of xfsg generated fault lists, only for SA faults. This script also allows a custom fault list to be given as an input such as follows:

```
./xfs_flow.sh serial custom
```

In this case, the serial flow will be instrumented, while elaborating all possible faults. However, fault injection will be done only on the faults provided in the custom fault injection tcl file inside the script. If no arguments are provided with the script, serial fault simulation on all possible faults will be run.

For **VC Z01X**, the concurrent engine is the default fault simulator, which is what we use to test the tool as well. The flow discussed in Section 4.1.2 is put in the form of a script to automate the flow. Options of running either the SA or transient faults are also provided along with the script. VC Z01X, by default, does not support classification of faults based on functional and checker strobes. Therefore, a custom SystemVerilog file (`strobe.sv`) for strobing purposes is developed in order to accommodate the same. Further, custom fault statuses are defined to incorporate the ISO 26262 defined fault classes.

In the `strobe.sv` file, the observation and detection points defined in the SFF file are strobed at the rising edge of every clock, with the help of `$fs_compare`, `$fs_observe` and/or `$fs_detect` system functions. Based on the result of the function, the corresponding fault is elevated to the desired status. For example, if there is a value mismatch for a checker strobe, the checker classification will be updated to Detected. Different possible combinations are taken care of in the strobe file to accommodate all possible scenarios. Fault campaigns are generated individually for SA and transient faults, with the reports merged together at the end.

The different configurations of both the tools will be tested on reference designs to check for any discrepancies in results, simulation times and any possible optimisations.

4.3.2. Full Adder

Figure 4.4 shows the design of the full adder, along with names of the inputs, outputs and the intermediate wires. Typically, functional outputs of the design are configured as functional strobes, while outputs from Safety Mechanisms are taken as checker strobes. Given the absence of a Safety Mechanism for this simple design, we opt to designate S and C_{out} as functional and checker strobes respectively. Fault injection is performed on all possible locations in the design, as marked in Figure 4.4.

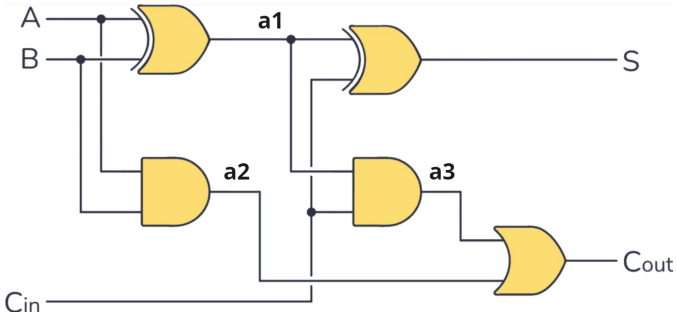


Figure 4.4: Binary Full adder

For the verification setup, we exercise all possible combinations of inputs one by one in a single test, and observe the outputs. Fault classifications are then made based on the differences in the output values of good and faulty simulations.

4.3.3. 4-bit up counter

Listing 4.1 shows the design of a simple 4-bit upcounter design. With the first design being a combinational circuit, this setup allows us to analyze the results on sequential elements. Fault injection will be performed on all possible locations in the design, including internal variables in RTL. `counter[3:2]` is set as the functional output, while `counter[1:0]` is designated as the checker strobe. The verification setup includes setting up a clock (period of 10ns), a reset, and letting the simulation continue until a rollover of the counter has occured (170 ns).

```
1  `timescale 1ns/10ps
2  module upcounter(input clk, reset, output[3:0] counter
3      );
4    reg [3:0] counter_up;
5
6    //up counter
7    always @(posedge clk or negedge reset)
8    begin
9    if(~reset)
10    counter_up <= 4'd0;
11   else
12    counter_up <= counter_up + 4'd1;
13   end
14   assign counter = counter_up;
15
16 endmodule
```

Listing 4.1: Up counter verilog design

4.3.4. FIFO design with Safety Mechanisms

Figure 4.6 shows a top level overview of a FIFO design, equipped with ECC. This design is provided by Synopsys as a reference design for FuSa purposes. The FIFO_SM module contains an instance of a Simple Dual Port RAM (SDPRAM) wrapper - SDPRAM_TOP, which contains an instance of the dual port RAM, with the two ports denoted by L and R. The L port is used to write the data on to the FIFO, whereas the R port is used to read data from the FIFO. The signals not used on a particular port are thus left disconnected or tied to 0. The SDPRAM contains the memory array, which corresponds to the FIFO, and is configured with a depth of 8, with the data width of each element being 12(8 data bits + 4 bit ECC). The SDPRAM wrapper also contains an instance each of ECC encoder and decoder modules.

When data is written to L port after setting L_WriteEn to 1 and passing the data to L_DataIn, the data goes to the ECC encoder and gets encoded with 4 additional ECC bits to be stored in the FIFO at the location specified by L_Address. When data is read from the R port, by specifying the R_Address and enabling the R_ReadEn, the data from the FIFO is fed to the ECC decoder, which then decodes the data and passes the 8 bit data to the output. If there is an ECC error, the corresponding **EccError** signal is also triggered.

There are two additional modules, called the FLAGS and COUNTER. The FLAGS module takes the read and write enable signals to determine the status of the FIFO, whether it is Full, Half Full or Empty. As shown in Figure 4.6a, two instances of this module are enabled in FIFO_SM module, with one of them being redundant logic. The two outputs are compared in order to trigger a **FlagError** whenever applicable. The second module, COUNTER, either the read/write enable signals as inputs, in order to calculate the read/write pointer. As shown in Figure 4.6b, the COUNTER modules are duplicated for calculating write pointer, with the DoWrite signal being passed as an input to both the modules. If there is any difference between the outputs of the two modules, a **WriteError** is triggered. Similarly, for the read pointer as well, such a setup is implemented to eventually trigger a **ReadError** whenever applicable. The top level **Error** signal in SM_TOP is triggered when any one of EccError, ReadError, WriteError or FlagError is high. The **Error** signal is thus configured as a **checker** strobe for this design. On the other hand, the **DataOut** signal is taken as the **functional** strobe, along with the three flags - Empty, Full and Half Full.

The WriteEn and ReadEn signals in FIFO_SM are triggered from the testbench. Based on these signals and the value of flags, the corresponding DoWrite and DoRead signals are set. For example, DoWrite is set only when WriteEn is high and Full flag is not set. A similar logic follows for the DoRead signal as well.

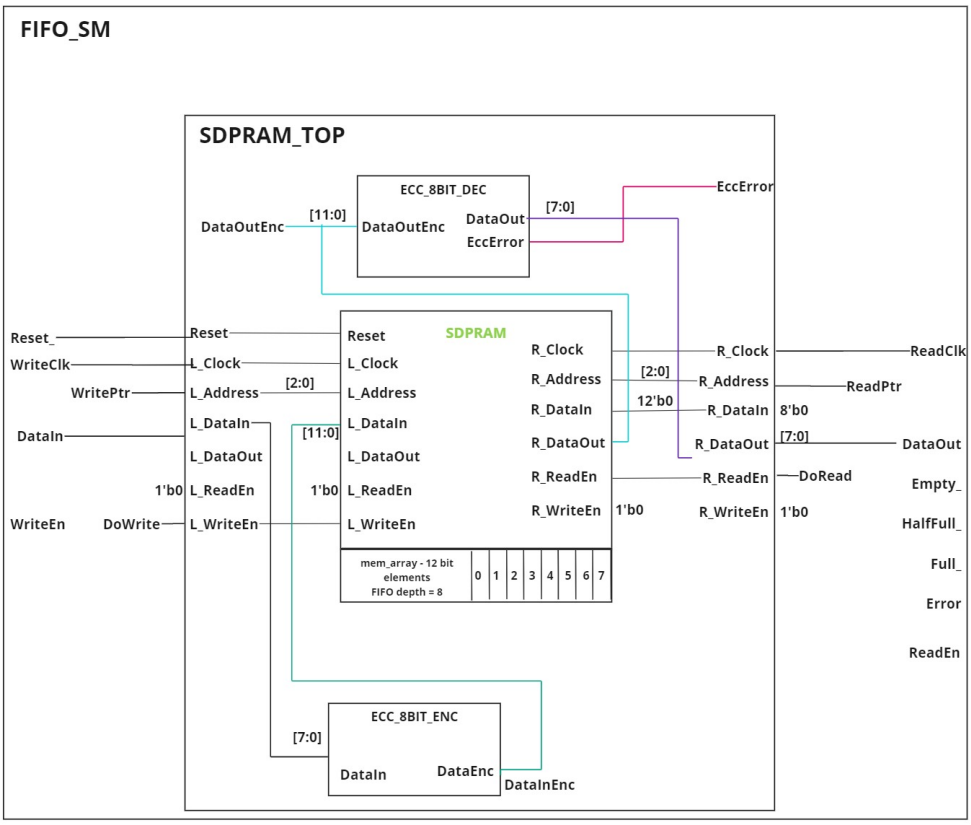
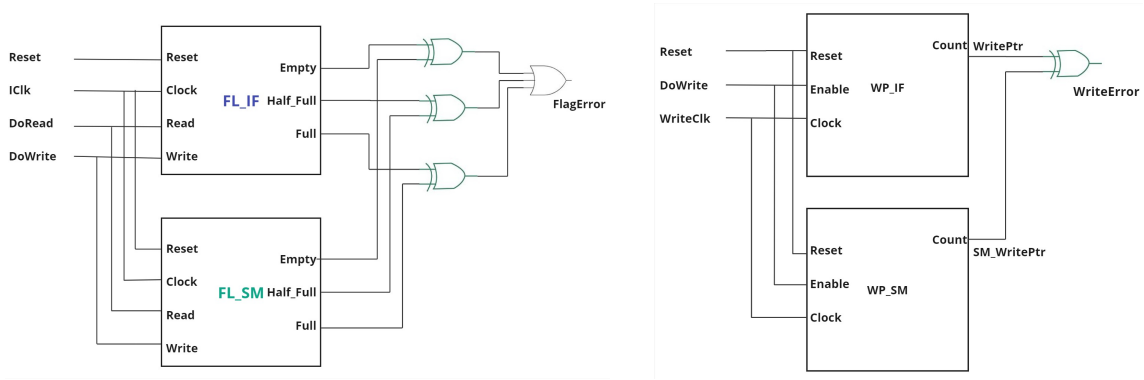


Figure 4.5: Top level diagram of FIFO with ECC



(a) Flags module

(b) Counter module

Figure 4.6: Redundant architecture of modules for error detection

On the verification side, a clock with a period of 100 ns is set up, and the reset is initiated after 220 ns. Functions are defined to automate write and read tasks, along with another function to manually check the status of flags. A couple of tests are defined, each comprising a sequence of Read/Write Operations to test the FIFO operations.

In the next section, we discuss the results of using the automated tool flows on these reference designs, followed by drawing comparisons according to the defined metrics.

4.4. Tool outcomes and Comparison

In the first step, we run the three versions of XFS flow, as defined in Section 4.3.1, on the adder design to look at the results. To recap, faults detected at the functional strobe are classified as **Observed**; if not, they are labeled as **Not Observed**. Similarly, faults identified at the checker strobe are classified

as **Diagnosed** or **Not Diagnosed**. In the context of FuSa verification, the goal is to reduce the number of **Observed Not Diagnosed** faults. These faults indicate that the functional output of the design is affected, and the Safety Mechanism fails to detect them. Minimizing these types of faults contributes to enhancing the Diagnostic Coverage, potentially resulting in a higher ASIL level.

Table 4.1 shows the results of Stuck-At faults on the adder design. There are a total of 8 faultable locations, each being injected with SA0 and SA1 fault, leading to a total of 16 faults. "Serial random" strategy corresponds to the faults being run serially with random fault ID. "Serial xfsg" also refers to serial simulation, but with fault injection through a tcl file generated with the help of xfsg. "Mixed concurrent" strategy refers to running the SA faults of the design with the help of concurrent engine, and simulating the SET faults with serial engine.

Table 4.1: Results of Stuck-At faults from 3 flows of XFS(Adder)

Classification	**Serial random**	**Serial xfsg**	**Mixed concurrent**
Not Observed Diagnosed	6	6	4
Not Observed Not Diagnosed	0	0	0
Observed Diagnosed	8	8	8
Observed Not Diagnosed	2	2	2
Not Simulatable	0	0	2
Total	16	16	16

As seen from the table, the results of "serial random" and "serial xfsg" are same, and in line with the manual analysis of fault results done on the adder design. However, with the concurrent flow, we see that 2 of the faults which are supposed to be **Not Observed Diagnosed** are classified as **Not Simulatable (NS)**. These two faults correspond to the location of the internal wires, **a2** and **a3**. However, wire **a1** is still being simulated by XFS correctly in the concurrent engine. Faults are usually classified as NS by XFS if they are not supported by the concurrent engine. This shows that there is a discrepancy in fault injection on intermediate nodes, as Xcelium seems to optimize the two wires a2 and a3 for SA1 fault, and thus, not consider them for simulation. Such a behaviour is not seen with the serial engine. In order to avoid such an optimization, a new switch `-nogateamalg` needs to be provided. Upon providing the given switch, the results of the three flows for SA faults come out to be the same.

For transient space, the faults are injected at random times in the simulation on all 8 fault locations. As seen in Table 4.2, the fault classification numbers for the serial random simulation differ from that of SA faults. This is due to the fact that the transient faults are injected at random times. However, for the XFSG generated fault list with default options provided by the utility, none of the faults are injected on the first go. Thus, in order to fix this, fault injection times need to be manually put in the fault injection tcl file. With the updated tcl script, the fault classification results fall into the other 4 categories in the same way as the "serial random" strategy, provided that the fault injection times are same for both the cases.

Table 4.2: Results of Transient faults from 2 flows of XFS (Adder)

Classification	**Serial random**	**Serial xfsg**
Not Observed Diagnosed	3	0
Not Observed Not Diagnosed	0	0
Observed Diagnosed	2	0
Observed Not Diagnosed	3	0
Not Injected	0	8
Total	8	8

There is also a single-step mode, wherein all the steps mentioned in the XFS flow can be run with the help of a single command. The user can provide the design and testbench files along with the fault specification target, strobe list, and specify the fault engine to be used - serial or concurrent, in a single invocation of the xrun command. This would facilitate ease of usage for the user. However, the tool internally uses xfsg for this flow, and with xfsg not providing hold time for SET faults, they are not injected. Further, there seems to be an issue when both the functional and checker strobes are

mentioned together in a single line, leading to difference in classification results. This flow is thus not considered for further discussion.

The **VC Z01X flow** is instrumented on the adder flow, with the same fault locations as targets. VC Z01X also provides options for the location type (PORT, PRIM, FLOP, ARRY, WIRE, VARI), as mentioned earlier. So, for the same fault specification file, VC Z01X instruments more number of faults. By default, if all the location types are enabled, different types of faults are generated for the same fault location, as also seen in Figure 4.7 (XFS results on the left versus VC Z01X results on the right). If we take the example of port Ain, we see that that there are two faults generated in XFS, but 4 faults in case of VC Z01X. This is because the signal Ain is instrumented both as a port and a wire in VC Z01X. In this example, the classification does not change whether we inject the fault on the port/wire. However, as we will see in further designs, the classification might change depending on whether a fault is injected on a wire connecting the port or the port itself.

Similarly, for the port Cout, we see that there are 3 types of locations being exercised - port, wire and primitive. **The "- -" in the VC Z01X report denotes that the signal is part of the collapsed list of signals, and follows the same classification as the first signal in the collapsed list.** The fault classifications of the signals do not change necessarily in this example, as compared to the XFS results. However, there is a big variation in the number of faults being generated for each tool, as also illustrated in Table 4.3. The first column represents the SA fault results of XFS, with the second and third columns providing VC Z01X SA fault classifications. The second column contains the complete list of faults instrumented at all possible location types. The final column contains the results for faults injected only at wires. Upon manual inspection of all the signals, the fault classification comes out to be the same for all signals exercised on both tools.

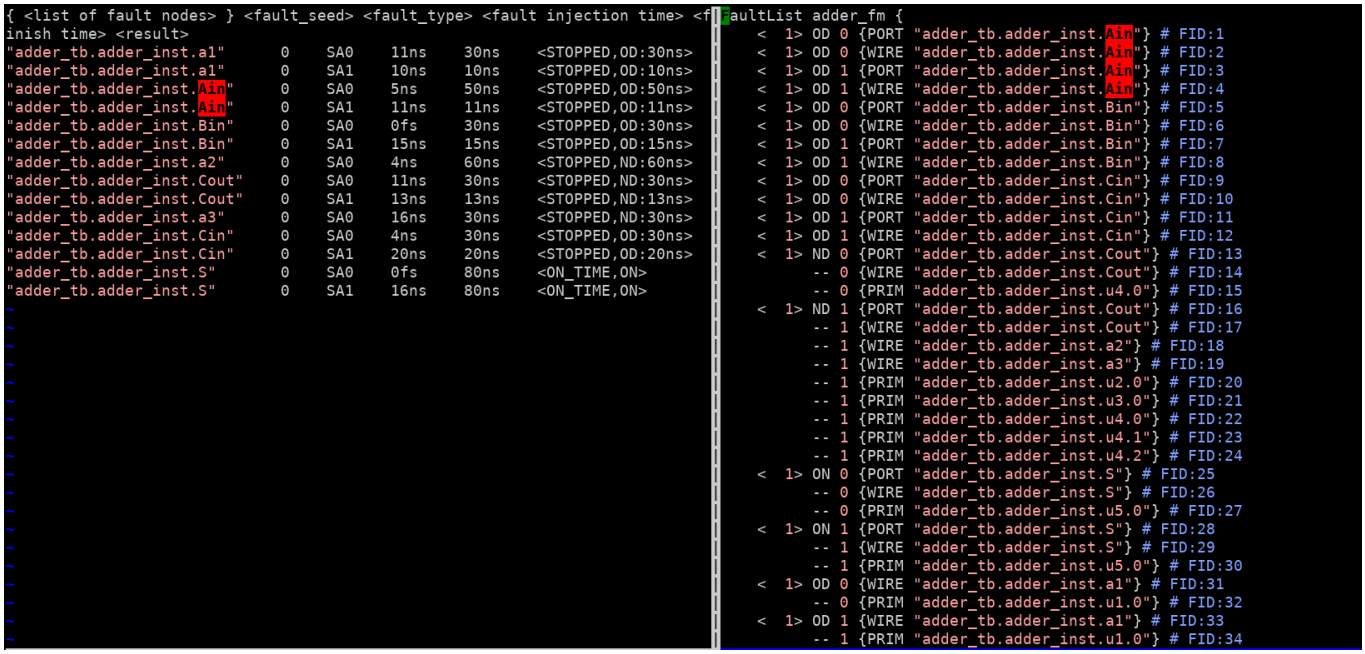


Figure 4.7: Comparison of faults generated with both tools (XFS vs VC Z01X)(Adder)

Table 4.3: Results of Stuck-At faults (Adder) of XFS and VC Z01X

Classification	XFS	VC Z01X default	VC Z01X compressed
Not Observed Diagnosed	6	26	6
Not Observed Not Diagnosed	0	0	0
Observed Diagnosed	8	20	8
Observed Not Diagnosed	2	10	2
Total	16	56	16

Results for SA faults for the counter design is presented in Table 4.4. As seen earlier with the adder

design, the faults generated on the counter design with default setting of VC Z01X is more than that of the faults generated with XFS. However, in terms of classification, there is no discrepancy in the results obtained, and is consistent with the manual analysis performed on the design points.

Table 4.4: Results of Stuck-At faults (Counter) of XFS and VC Z01X

Classification	XFS	VC Z01X default	VC Z01X compressed
Not Observed Diagnosed	4	8	4
Not Observed Not Diagnosed	0	0	0
Observed Diagnosed	8	16	8
Observed Not Diagnosed	8	16	8
Total	20	40	20

In evaluating the transient fault classification for both designs, we find that the outcomes are consistent with the faults injected at various design points over time. However, due to the random timing of fault injections in both tools, comparing the results becomes challenging. Furthermore, considering the discrepancy in the number of faults generated by both tools, we first try to create a common ground for the tools in terms of the fault space and then analyze the results. This is the approach we adopt first for the FIFO RTL, a design which is also relevant for FuSa, as it contains Safety Mechanisms for fault detection. We will also look at a way to make the transient fault lists similar for both the tools in order to aid comparison.

SA faults are instrumented on the FIFO design only on ports to keep the fault space same for the tools. This allows a common ground for comparison, and can be built upon for the extended fault list. The classification results for the FIFO are depicted in Table 4.5, highlighting major discrepancies in the classification of faults.

Table 4.5: FIFO results for SA faults on ports - XFS v/s VC Z01X

Classification	XFS	VC Z01X
Not Observed Diagnosed (ND)	77	94
Not Observed Not Diagnosed (NN)	98	1
Observed Diagnosed (OD)	119	74
Observed Not Diagnosed (ON)	84	112
Total	**378**	**281**
Untestable/Safe faults (UU)	44	88
Not Controllable (NC)	- -	53
Total	**422**	**422**

As seen above, the **Untestable/Safe (UU)** faults classification encompasses all faults identified as Safe by each tool. Further, VC Z01X provides a **Not Controllable (NC)** fault class, which denotes signals that do not toggle during the good simulation, and hence a fault injected with the same value is an NC fault. The different classifications from both the tools are extracted with the help of a Python script and discussed further in the next section.

4.4.1. Automated comparison scripts

While it was straightforward to manually check the classifications for a simple adder or counter design, the comparison becomes much more daunting on a more complex design such as FIFO. A script is developed in order to check the final reports of both the tools and provide three outputs -

- The first file (`comp_same_final.txt`) contains all the signals whose classifications match for both the tools.
- The second file (`comp_diff_final.txt`) contains the signals whose classifications do not match.
- The final file (`comp_not_found.txt`) contains any signal which is found in one tool, but not found in another.

The classification names for both the tools are different, and hence a simple search and replace is performed in order to convert a particular classification of one tool to another (for example, Dangerous Detected in XFS to Observed Diagnosed in VC Z01X). This is done for different fault classes supported by XFS and converted to the corresponding classification in VC Z01X.

Given that VC Z01X had a more extensive set of faults instrumented, each signal in this report was individually parsed using regular expressions (**regex**), along with the fault type and fault ID (FID). Subsequently, each regex match was cross-referenced with the corresponding signal and fault type in the XFS report, and the classification was checked. Based on whether the classifications matched or even existed, the signals with their respective classifications were categorized into one of three output files. The FID was also included in the final report to facilitate quick debugging in VC Z01X using the FID. The test used along with information about collapsed signals is also presented in the report. It is important to note that the script did not consider location type, as XFS does not distinguish location type in its final report. As shown in Figure 4.8, `comp_diff_final.txt` contains the fault classification result from both the tools, the fault type, the signal name, FID from VC Z01X, test used for this fault, and information about collapsed signal. Such a comparative analysis script could be extended for more tools, if required.

```
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.L_WriteEn, FID: 370, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[0], VC-FID: 373, TestNum: 1
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.R_Address[0], FID: 374, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[0], VC-FID: 375, TestNum: 1
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.R_Address[0], FID: 376, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[1], VC-FID: 377, TestNum: 2
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.R_Address[1], FID: 378, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[1], VC-FID: 379, TestNum: 1
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.R_Address[1], FID: 380, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[2], VC-FID: 381, TestNum: 1
XFS: OD, VCZ01X: ON Fault_type: SA0 Signal: test.DUT.sdpram_i1.R_Address[2], FID: 382, Collapsed signal
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.sdpram_i1.R_Address[2], VC-FID: 383, TestNum: 2
XFS: OD, VCZ01X: ON Fault_type: SA1 Signal: test.DUT.sdpram_i1.R_Address[2], FID: 384, Collapsed signal

XFS: ON, VCZ01X: OD Fault_type: SA1 Signal: test.DUT.FL_IF.Write, VC-FID: 66, TestNum: 2
XFS: ON, VCZ01X: OD Fault_type: SA0 Signal: test.DUT.WP_IF.Clock, VC-FID: 115, TestNum: 1
XFS: ON, VCZ01X: OD Fault_type: SA1 Signal: test.DUT.WP_IF.Enable, VC-FID: 124, TestNum: 1

XFS: OD, VCZ01X: ND Fault_type: SA0 Signal: test.DUT.FL_SM.Clock, VC-FID: 67, TestNum: 1
XFS: OD, VCZ01X: ND Fault_type: SA1 Signal: test.DUT.FL_SM.Clock, VC-FID: 68, TestNum: 2
XFS: OD, VCZ01X: ND Fault_type: SA0 Signal: test.DUT.FL_SM.Read, VC-FID: 75, TestNum: 2
XFS: OD, VCZ01X: ND Fault_type: SA1 Signal: test.DUT.FL_SM.Read, VC-FID: 76, TestNum: 1
XFS: OD, VCZ01X: ND Fault_type: SA0 Signal: test.DUT.FL_SM.Reset_, VC-FID: 77, TestNum: 1
XFS: OD, VCZ01X: ND Fault_type: SA0 Signal: test.DUT.FL_SM.Write, VC-FID: 79, TestNum: 1
XFS: OD, VCZ01X: ND Fault_type: SA0 Signal: test.DUT.RP_SM.Clock, VC-FID: 97, TestNum: 2
XFS: OD, VCZ01X: ND Fault_type: SA1 Signal: test.DUT.RP_SM.Clock, VC-FID: 98, TestNum: 1
```

Figure 4.8: Differences in classification results extracted from script

4.4.2. Analysis of Stuck-At Fault classification differences of FIFO

As shown in Table 4.5, the first major difference we see is in the classification of NN faults for XFS and VC Z01X. NN faults usually demand manual analysis to determine whether the fault is genuinely safe or if it remains undetected due to limitations in the test. The difference of 97 faults is due to the 44 UU and 53 NC faults detected by VC Z01X. XFS and VC are both able to detect 44 faults in common, but VC goes a step further in determining 97 additional faults into two separate categories. This saves the user in spending additional time to debug the classification of NN faults. UU faults can be considered Safe, whereas the NC faults can be parsed to appropriately toggle the signals in a new test case. The classifications provided by VC Z01X play a crucial role in reducing manual effort in fault analysis. Moreover, they offer insights into the location of NC faults, aiding in the development of more effective tests to detect faults at strobing points.

The 44 faults which are deemed Untestable by VC Z01X and NN by XFS contain the signals L_ReadEn, R_WriteEn and R_DataIn signals, all of which are constrained to 0 and left unused on both ports. The L_DataOut signals are classified as Untestable in both the tools, as no connection is made to this port. The 53 faults classified as NC by VC Z01X and NN by XFS consists of Reset signals in all hierarchies and the DataOut signals. All faults are injected after the Reset is asserted, and hence the SA1 faults on Reset are not controllable. Similarly, because the data written is targeted rather than random, not all data bits are toggled, resulting in some bits of the DataOut signals being

uncontrollable. While the Reset signal classifications can be deemed Safe, the DataOut signals can undergo testing with a variety of randomized data inputs to classify them as testable.

The next set of faults consists of signals which are classified as **OD by XFS, and ON by VC Z01X.** Output signals - Empty, Full, HalfFull and input signals such as L_Address and R_Address fall into this category of classifications. In order to better understand the cause of this difference, let us take the example of a **SA0** fault on the output port **Empty**. According to the behaviour of both tools as mentioned in the user guides, a fault injected at an output port will be placed on all the external loads connected to that output port outside of the module. Similarly, a fault injected on an input port will be placed on the internal loads connected to the port within the module and hierarchy below it. This is taken care of by a method called "fault isolation", wherein pseudo buffers are placed at the ports in order to avoid unnecessary propagation.

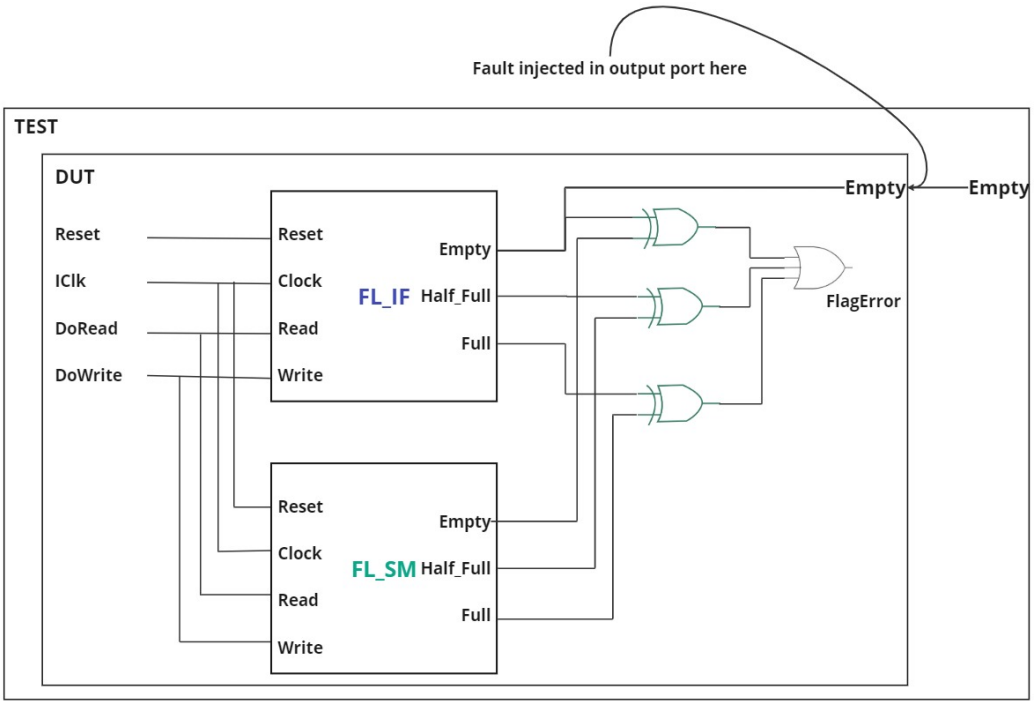


Figure 4.9: Fault injected on DUT output port Empty

According to fault isolation rules, a fault injected on the output port Empty of the DUT, as shown in Figure 4.9, should propagate outwards towards the Empty port of test module. However, as seen in a fault simulation run of XFS (SA0 fault on test.DUT.Empty_), there is a back propagation transmitted to the Empty_ wire of the DUT module, as shown in Figure 4.10. Hence, test.DUT.FL_IF.Empty_ is also stuck at 0 for the simulation. This leads to a difference in the Empty_ flags of the FL_IF and FL_SM modules (which is still behaving correctly as expected), leading to a FlagError, and consequently a detection at the checker point. The Empty_ signal is already considered as a functional strobe, and hence the classification provided by XFS is OD. On the other hand, VC Z01X, does not back propagate the fault and therefore, it is not observed at the checker output. Thus, the classification of the same fault comes as ON, and hence the difference between the two tools.

Similarly, for a fault injected at the input port - L_Address of SDPRAM_TOP, as shown in Figure 4.11, the fault is propagated backwards in the direction of WritePtr, which is also stuck at the given value now. The WritePtr for the SM module is still behaving correctly, leading to a difference in the two WritePtr signals, and consequently a detection in the WriteError signal. Hence, the classification for this set of signals is also OD, as provided by XFS, and ON by VC Z01X.

Another set of signals, specifically inputs to the redundant modules of flags, read and write pointers, are classified as OD by XFS, and ND by VC Z01X. Taking the example of a fault injected at "test.DUT.FL_SM.Write", as shown in Figure 4.12, a fault injected at input port Write, should be exercised on all internal loads within the modules and hierarchies below it. However, the fault value

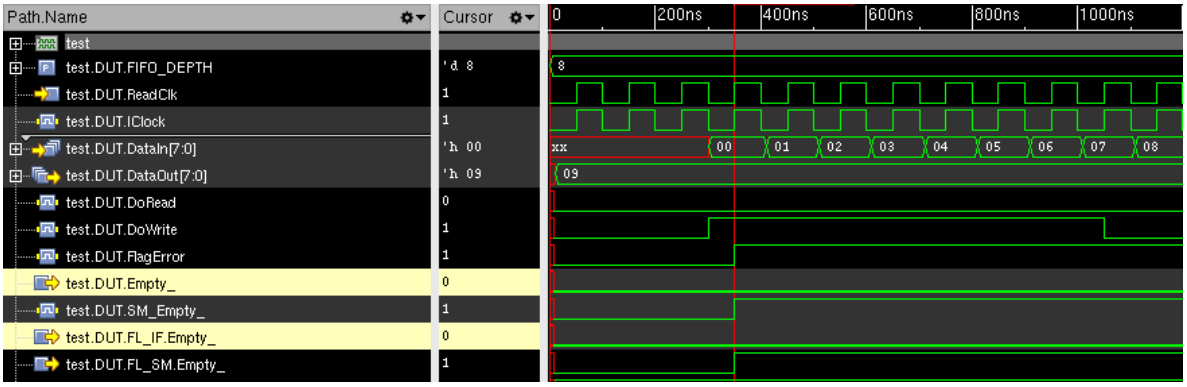


Figure 4.10: Back propagation of fault injected at Empty port

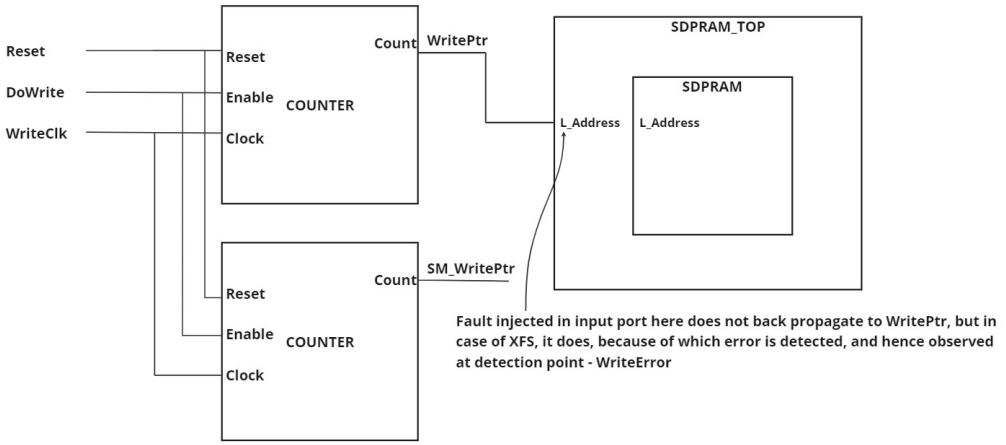


Figure 4.11: Back propagation for L_Address signal in SDPRAM TOP

is also seen at "test.DUT.FL_IF.Write" because of back propagation. This in turn is also seen at the functional output, and hence there is a difference in fault class at the functional output of both the tools. The same behaviour is observed for all input signals which are common to the SM modules.

To summarise, all these aforementioned differences in classification are due to the fact that fault isolation is not executed properly on the input and output ports of the RTL design by XFS, leading to back propagation of faults. In technical terms, the behavior of the fault in XFS is not inherently incorrect. Rather, the fault's modeling differs from what is expected. It also needs to be observed, after expanding the fault space to all possible locations, whether there is a distinct fault instrumented on the wire connecting the port. This scenario would be ideal for observing the effects of both faults on the design.

In the next step, we include all possible fault locations in the design and then compare the results of both the tools. The results of extending the Stuck-At fault space are illustrated in Table 4.6. As expected, the fault numbers are much higher for VC Z01X, because of the various fault location types provided by the tool. The comparison scripts are also used here for the two reports generated with this extended fault space.

The `comp_diff_final.txt` does not contain any new differences in the signal classifications other than the ones which are deemed NN by XFS and UU/NC by VC Z01X, arising because of new signal additions to these lists. However, since VC Z01X offers fault injection on the same signal with different location types, similarities are observed for both the tools with the classifications arising from such situations. For example, while instrumenting port fault on "test.DUT.Empty_", the fault classifications were OD and ON for XFS and VC Z01X respectively. However, the latter also instruments the same signal as a wire, in which case, the classification turns out to be OD, as expected and shown in

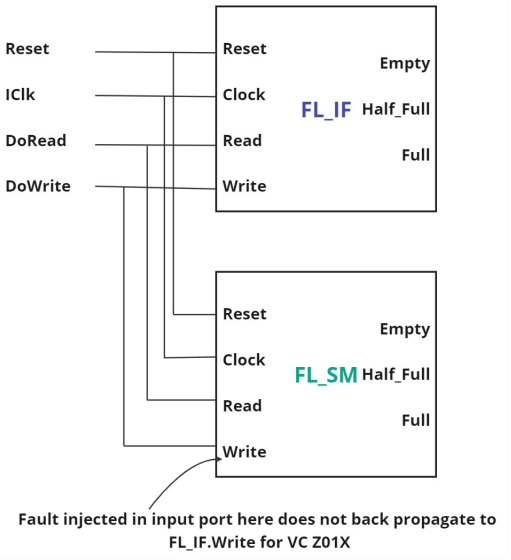


Figure 4.12: Fault injected on inputs of redundant modules

Table 4.6: FIFO results for SA faults on extended fault space - XFS v/s VC Z01X

Classification	XFS	VC Z01X
Not Observed Diagnosed (ND)	232	487
Not Observed Not Diagnosed (NN)	186	61
Observed Diagnosed (OD)	156	261
Observed Not Diagnosed (ON)	88	200
Total	**662**	**1009**
Untestable/Safe faults (UU)	44	176
Not Controllable (NC)	- -	223
Total	**706**	**1408**

Figure 4.13. **On the other hand, XFS does not inject these two faults separately and thus, we miss out on the effect of a particular fault.** In essence, for the extended fault list, we do not see any new irregularities in terms of fault classes provided by both tools, apart from the injection of faults at different location types in VC Z01X, which might lead to different results as shown above.

```
<  1> ON 0 {PORT "test.DUT.Empty_"} # FID:151
<  2> OD 0 {WIRE "test.DUT.Empty_"} # FID:152
      -- 0 {PORT "test.DUT.FL_IF.Empty_"} # FID:153
      -- 0 {WIRE "test.DUT.FL_IF.Empty_"} # FID:154
<  2> ON 1 {PORT "test.DUT.Empty_"} # FID:155
<  1> OD 1 {WIRE "test.DUT.Empty_"} # FID:156
      -- 1 {PORT "test.DUT.FL_IF.Empty_"} # FID:157
      -- 1 {WIRE "test.DUT.FL_IF.Empty_"} # FID:158
```

Figure 4.13: Fault injection on different location types in VC Z01X

Moreover, the `comp_not_found.txt` file now includes specific signals. This is a result of certain signals in XFS being excluded due to fault collapsing optimizations, resulting in their omission. However, all signals are included in the final report of VC Z01X, whether they are collapsed or prime faults, along with the fault classification. With this, we conclude with the results of the SA fault classification of both the tools for the FIFO design, with the main takeaway being the ***back propagation of faults in XFS***. In the next section, we look at the transient fault space of both the tools on the same design and analyze the results.

4.4.3. Transient fault space modelling for FIFO

XFS supports Single Event Transient (SET) with a hold time on possible fault locations, and Single Event Upset (SEU) on outputs of sequential elements. On the other hand, VC Z01X supports transient faults only on outputs and on location types - PORT, VARI, FLOP and ARRY. **This means that transient faults will not be injected on inputs and intermediate wires in VC Z01X.** XFS, however supports transient fault placement on all signals. Further, in order to inject a transient fault in XFS, one has to write a fault injection tcl file with the fault type, location and hold time for SET faults. On the other hand, VC Z01X provides options in the SFF file to create an extensive transient fault list. There is the option of a transient toggle fault, which can be specified with the help of a simple statement inside the FaultGenerate section such as :

```
NA ~ (3:13) {[PORT,ARRY,VARI,FLOP] "test.DUT.**" }
```
(4.2)

This means that transient toggle faults will be injected individually at cycles 3^{rd}, 4^{th}, 5^{th} and so on, until the 13^{th} clock cycle, on all possible locations supported in the design. There is also the possibility of a transient hold fault, which can be generated as :

```
NA ~ (8^11) {[PORT,ARRY,VARI,FLOP] "test.DUT.**" }
```
(4.3)

This means that a hold fault will be generated starting from 8^{th} cycle till 11^{th} cycle, on all possible locations in the mentioned hierarchy. The behaviour of these faults have already been discussed in Figure 4.3. With the options for transient faults being different in both the tools, we try to find a common ground to inject the faults with the two tools by modeling the transient faults in a particular way.

First, we run the transient faults with VC Z01X, with all possible options. From the output report generated, we try to model faults in the same way for XFS. For instance, let us consider the example shown below:

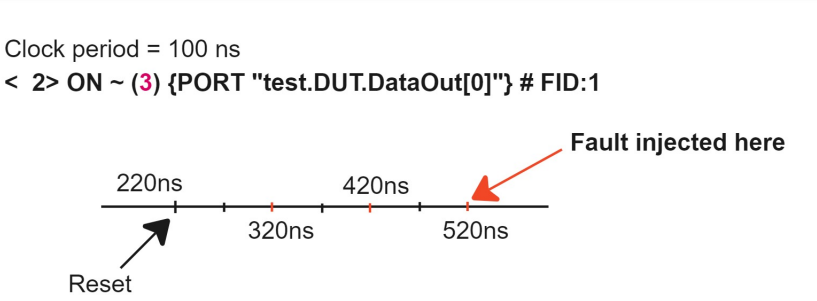


Figure 4.14: Transient fault injection in VC Z01X

As introduced earlier, VC Z01X has the option of specifying a cycle, with a period, in the SFF file. In this example, we assume a period of 100ns, which is the same as the clock period. However, the cycle considered by VC Z01X starts only after fault injection has been triggered (which is by default at time 0). Here, `$fs_inject`, which is the cue to start fault injection, is triggered only after reset has been asserted at 220 ns. Thus, all cycles start from that point onwards, as shown in Figure 4.14. So, if a fault is injected on the 3rd cycle, it will be injected at the end of the 3rd cycle starting from 220ns, i.e. 520 ns. The faulty signal will hold its value until the next clock edge comes at 550 ns, and based on the input, the signal will change its value. Similarly for transient hold faults, the signals will hold their value until the specified cycle.

Keeping in mind the behaviour of VC Z01X for transient faults, fault injection commands are generated by parsing every output from the VC report and subsequently injected in XFS. FLOP faults in VC Z01X are modeled as SEU faults in XFS with the corresponding injection time. All other faults are modeled as SETs, with the injection time and hold time set up as described before. Now that we have a common ground for both the tools, it becomes a little bit easier to run the comparison script on the two reports. As seen in the final classification results of the transient fault space in Table 4.7, there still seems to be a discrepancy in the final numbers, even though we have created the fault injection

tcl file from one of the tool reports. This is due to the fact that VC Z01X has different location types for a particular signal, because of which fault injection commands will be duplicated in XFS. These duplicated results are converted to one final classification in the final report generated by XFS. We also see one additional classification of Impossible X-state (IX), which refers to transient faults injected on signals which were in an Unknown state (x).

Table 4.7: Transient fault results for FIFO - XFS v/s VC Z01X

Classification	**XFS**	**VC Z01X**
Not Observed Diagnosed (ND)	1889	1909
Not Observed Not Diagnosed (NN)	637	815
Observed Diagnosed (OD)	248	318
Observed Not Diagnosed (ON)	390	474
Total	**3164**	**3516**
Impossible x-state (IX)	48	48
Total	**3212**	**3564**

First things first, the issues and differences which were already seen in stuck-at results of both the tools (specifically back propagation) are prevalent in the transient fault space as well. In addition to this, there are a few more differences seen in the results of the two tools. This is attributed to the fact that the strobing mechanism in both the tools is different, which makes a difference in case of transient fault behaviour. In VC Z01X, the observation and detection points are being compared at the positive edge of every clock, whereas in XFS, the strobing points are compared at every time step. Thus, if a transient fault is injected at a time just before the clock's rising edge (for example, at 1230 ns), and the effect of the fault is only for 20ns (till 1250ns), the effect of the fault will never be seen in VC Z01X, because the observation point is not compared until 1250 ns. However, in XFS, since the strobes are compared at every time step, the fault effect will be observed. Essentially, these differences arise from the strobing behavior and not from any issue in the tool itself. The strobing mechanism is dependent on the user and the requirements of the system, and can be configured accordingly.

To summarise the results of the transient fault space, there are no additional issues seen in terms of tool results, apart from the back propagation effect already seen in the stuck-at fault space. However, there are differences in fault space and modeling between the two tools, making direct comparison quite challenging. XFS has the capability to cover a larger fault space, as VC Z01X cannot generate transient faults on inputs and intermediate nodes. This factor must be considered in developing the final solution/verification flow. In the next section, we conclude with the results of these reference designs in terms of the metrics and set the stage for developing the final methodology.

4.4.4. Metric-based comparison and conclusions

In this section, we look at the results on the basis of metrics defined in Section 4.2, illustrated below:

- **Quality of results:** XFS has internal discrepancies in tool results with small reference designs. It is seen that the results from different flows are also not equivalent with each other at times. While most of these issues can be solved manually with additional configurations or switches, it still raises questions on the quality of tool results. On the other hand, VC Z01X does not have any such issues with the tool internally.

  One significant observation from the FIFO design results is the back propagation issue identified in XFS. While the fault behavior technically is not incorrect, XFS does not instrument port faults as expected. It is essential to distinguish between faults injected at ports and those affecting the wires connecting the ports because their behaviors may differ, as evidenced by the results from VC Z01X. On the other hand, VC Z01X lacks the capability to inject transient faults on inputs and intermediate wires, leaving a portion of the fault space uncovered. Although this is not a tool issue per se, achieving accurate fault coverage results may be affected if the entire fault space is not considered.

  In summary, when it comes to providing precise and accurate results, and accounting for fault behaviors across various location types, VC Z01X fares better than XFS in terms of result quality.

The only limitation is the portion of the transient fault space that remains unaddressed by the tool.

- **Comparison of features:** A detailed comparison of the different features provided by the two tools is tabulated in Table 4.8. The comparison covers aspects such as fault classifications, fault models, specification files, fault strobing and simulation options among others. Similarities and differences are analyzed between these different features in order to get a deeper understanding of the tools.
- **Run-time comparison**: The comparison of run-times for each of the three designs with both tool flows is illustrated in Figure 4.15. The XFS flow implemented here utilizes a serial engine with an xfsg-generated fault list. This choice was made over the concurrent engine due to the limitations of the latter in instrumenting SET faults and limitations of several constructs and HDLs.

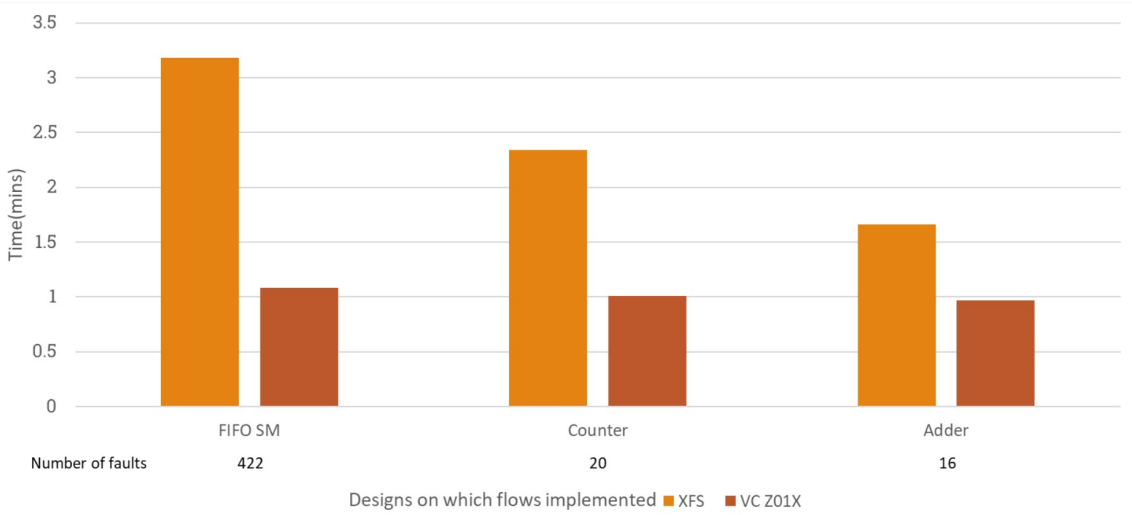


Figure 4.15: Run time comparison of the tools on reference designs

As seen in the comparison, VC Z01X is faster than XFS in all flows tested. This advantage is primarily attributed to the concurrent engine employed by VC Z01X. While XFS does offer a concurrent flow, it presents limitations such as incompatibility with RTL designs, unsupported constructs, and the inability to work with VHDL and System Verilog designs. Consequently, VC Z01X exhibits better scalability in terms of run-time, particularly for larger and more complex designs, as it can instrument thousands of faults in one shot. As depicted in Figure 4.15, the run-time for the VC Z01X flow does not increase significantly across the three designs compared to the XFS flow. In the XFS flow, each fault must be injected individually, resulting in longer simulation times for larger designs. Although scripts can be developed to parallelize fault injection lists and run them concurrently, this approach requires multiple tool licenses as well.

- **Features for performance optimisation**: Concurrent simulation is the primary feature which helps in improving simulation times. Concurrent engine simulates thousands of faults simultaneously, speeding up fault simulation times by a large factor. This capability is particularly advantageous for larger designs, where the impact is more visible. Both XFS and VC Z01X supports concurrent simulation, with the latter supporting the concurrent engine by default. XFS has limited concurrent support and has restrictions with respect to using different constructs in RTL design. XFS also has discrepancies corresponding to the instrumentation of intermediate wire faults while performing concurrent simulation. Additionally, since SET faults are not supported in XFS concurrent simulation, a serial run has to be triggered to cover the remaining fault space. This defeats the entire purpose of concurrent simulation, thus failing to achieve the expected improvements in simulation times characteristic of the concurrent engine.

  XFS also supports a construct in the strobing mechanism wherein simulation can be stopped when a fault is detected (as shown in Figure 4.16), either at a functional or a checker strobe point.

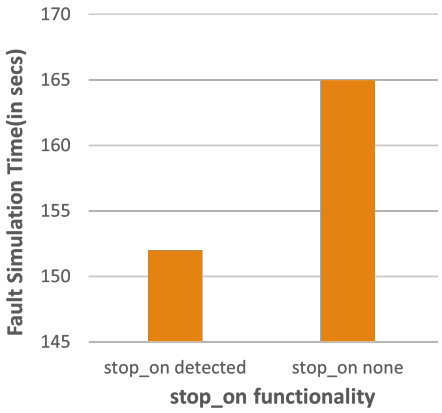


Figure 4.16: "stop_on" functionality in XFS strobing to reduce simulation time

This functionality can reduce simulation times, especially for longer tests, if faults are detected early in the test. VC Z01X, by default, supports this feature and stops simulation once faults are detected. However, simulations can be allowed to continue as well with the help of system calls in strobing file in a way similar to XFS. To summarise the results of features for performance optimisation, VC Z01X is much faster than XFS in terms of fault simulation times, mostly owing to the concurrent engine support being well-equipped and bug free. With all optimized settings as well, XFS completes fault simulation slower than VC Z01X.

Fault sampling is also an interesting feature which could be used to prune the fault space, consequently reducing fault simulation time. While this feature provides an estimate rather than an exact Diagnostic Coverage metric, it allows for substantial reduction in fault simulation time by reducing the number of faults and estimating the DC within a certain range. In the context of the FIFO example, fault sampling does not show any significant changes as the reduction in the number of faults is minimal. However, in more extensive designs, for example when addressing internal memories for CPUs, using fault sampling can reduce the overall fault space and save fault simulation time.

- **Feasibility of tool**: Among the three main tools initially considered for Functional Safety verification, Siemens' tool was excluded due to the unavailability of a license. VC Z01X and XFS emerged as viable options for our verification flow. However, XFS has several challenges with RTL constructs in the concurrent engine flow, leading to numerous faults classified as Not Simulatable (NS). Further, there are issues with back propagation on input and output ports, leading to different classification results as compared to VC Z01X. Despite these limitations, XFS is still feasible for usage in verification, particularly with the serial engine. The same feasibility holds true for VC Z01X. VC Z01X also fails to cover a portion of the transient fault space, in which case, XFS can come into the fore. In summary, while both VC Z01X and XFS can be incorporated into our verification flow, careful consideration of their specific capabilities and limitations is important to develop a final verification methodology.

- **Ease of usage**: Upon an extensive exploration of the user guides of the two tools and understanding the different options and features, it is quite straightforward to set up fault campaigns in both tools and automate them. However, different setups and prerequisites such as license paths, setting up environment variables and installation directories need to be performed before using the tools. In VC Z01X, majority of the information regarding fault simulation is kept in a single SFF file, facilitating easy usage. Fault list generation is a single step process in VC Z01X, specified in the SFF file, and taken care of by the Fault Campaign Compiler. In XFS, apart from specifying the fault specification file, one has to use xfsg in order to generate the fault injection files. Also, the fault specification and strobing files are separately created, and need to be changed according to the design in use. That being said, it is not a major drawback in terms of ease of usage. However, with SET faults not being generated with an automatic hold time, the user has to manually specify hold times for SET faults in XFS. These are some minor differences in the usage of both tools, but are not significant enough to dismiss either tool based on ease of usage.

Table 4.8: Comparison of individual features

Feature	**Cadence Xcelium Fault Simulator (XFS)**	**Synopsys VC Z01X**
Engines for fault simulation	• XFS has two engines - serial and concurrent, which can be used for fault injection campaigns. With serial fault simulation, the good simulation needs to be run first, followed by the faulty simulation. The concurrent engine runs both the serial and faulty simulations in one go. There is no default engine as such, and the user needs to specify with command line options as to what engine the fault campaign is to be run on.	• VC Z01X also has two engines like XFS, serial and concurrent. However, the default engine in this case is the concurrent engine. In order to invoke the serial engine for non-concurrent friendly faults, the serial fsim switch is to be added while elaborating the design and running fault simulation.
Stimulus	• Only testbench (SystemVerilog, Verilog, VHDL) is supported as a stimulus in XFS. The required testbench is to be provided during the elaboration stage of the XFS flow. VHDL and SystemVerilog are however not supported in the concurrent engine.	• VCZ01X supports Verilog and SystemVerilog(UVM) testbenches as stimulus. Additionally, there are other stimulus options, which includes FSDB (Fast Signal Data Base), VCD (Value Change Dump), eVCD. • This allows to reuse dumps from previous simulations in different fault simulation campaigns. For example, regressions run using Xcelium could be enabled to generate FSDB dumps, which can then be used in the Synopsys flow for fault campaigns.
Elaboration, compilation, good and faulty simulation setup	• In XFS, the first step in any FuSa flow is to elaborate the fault specification file (which contains the locations of fault instrumentation). This elaboration is performed with xrun, and has some additional options for the configuration of the fault database to be generated. There are options to disable fault isolation at ports of the design, to disable gate collapsing faults, etc. • The next steps in the flow are the good and faulty simulation runs, with the desired options triggered with xrun.	• In Synopsys, the first step is to compile and simulate the design. For VC Z01X, in this step, the fsim switch is added in order to enable fault simulation. However, while compiling the design, some additional options/args need to be provided so that faults at different locations are instrumented. For example, PORT, VARIABLE and FLOP faults instrumentation is not on by default, so additional switches need to be provided in order to instrument these faults. • VC Z01X takes care of good and faulty simulations in one go in the next step with the defined SFF file.

Fault classification	• The fault classifications in XFS are based on the functional and checker strobes (if present), or whether the faults are untestable, not simulatable, undetected. The priority of fault classification is as follows: Dangerous_Detected (DD or OD) > Dangerous_undetected (DU or ON)> Unobserved_detected (UD or ND) > Unobserved_undetected (UU or NN) , Detected > Potentially_detected > Untestable (S) > Undetected > Not_simulatable. • For a dual strobe list (with both functional and checker strobe points), the first four classifications correspond to the detection of functional and checker strobing points. For a single strobe list, Detected, Potentially_detected and Undetected are the corresponding classifications. Untestable and Not Simulatable faults do not depend on the strobing list.	• For Synopsys, the fault classification is much more detailed. The faults are first divided into different built-in status groups, such as Detected, Potential, Oscillating, Hyper, Illegal, Unselected, Untestable, Excluded and Not included in Built-in-status groups. These groups have different fault classifications inside them. We can also add custom fault classes in the Fault specification file in order to enhance the classifications if required. • The interaction between the different fault classes is governed by the Promotion Table. There is a default promotion table which shows the interaction between the current fault status, new fault status and the merged fault status for the built in statuses. If custom classes are added, then a new Promotion Table could also be applied in the fault specification file. • There is no separate distinction between the detection of Observation and Detection points in terms of the classifications made in VC Z01X. Custom classes need to be defined to accommodate the same, along with appropriate strobing mechanisms.
Fault models	• XFS supports SA 0/1 faults which can be applied to any kind of signal. Single Event Upset (SEU) does a bit flip on the output of a sequential element, and holds the modified value until it is assigned a new value. Single Event transient (SET) also does a bit flip, but holds the value for specified period of time, and can be applied to any kind of signal.	• VC Z01X supports SA 0/1 faults on all types of signals, and makes a distinction between the type of signal on which it injects the fault (e.g. wire or port). Transient faults are also supported, but only on outputs and not on primitives and wires. • There are two types of transient faults supported - transient toggle and transient hold, in a way similar to SEU and SET faults in XFS. VC Z01X also supports a special fault class, called Transition faults, which could be either a slow-to-rise or a slow-to-fall fault. This could be enabled with the help of additional args with fsim switch during elaboration, mentioning the transition delay of the fault. Although, it is not a requirement to instrument this class of faults for ISO 26262, it is a good feature to have.

Fault specification file	• The fault specification file in XFS consists of the targets for fault instrumentation, and can contain modules, instances or signals. It is composed of two types of commands, fault_target and fault_exclude. • fault_target is used to enable a particular target for fault instrumentation, and can take multiple options in order to instrument, say, net faults, port faults, faults on sequential elements, and most importantly the type of faults - SA0/1, SEU, SET or all of them. fault_exclude is used to exclude certain targets for fault instrumentation.	• The SFF file in Synopsys consists of constructs similar to fault_target and fault_exclude in order to include/exclude different modules, instances, etc. However, the fault instrumentation option also contains timing information for transient faults, and different location types for generation of faults. • The SFF file also contains the signals which are used for strobing purposes and additional sections such as Sampling, Coverage and FaultStatus definitions, among others.
Stuck-At-faults	• The Stuck-At-fault syntax for XFS is straightforward, wherein only the fault type is to be mentioned in the fault specification file. However, it lacks configurability in terms of the different location types where the fault can be instrumented. The location type (nets, ports, cell ports, sequential elements etc.) has to be provided through the elaboration option every time a specific fault needs to be instrumented, but does lack in options as compared to Synopsys tool.	• The Stuck-at-fault syntax for Z01X provides options with respect to location type. The fault type is mentioned first followed by the location info consisting of the location type(WIRE, PRIM, PORT, ARRY, FLOP or VARI) along with the path to the instance. This gives us more configurability with respect to fault location types and different fault behaviours. When required, some of the individual options can be turned off, which will be taken care of by the fault campaign manager.
Transient faults	• The two transient faults in XFS, SEU and SET, do not have extended options for configurability both in terms of location types and the timing options that go along with it. The SEU faults can be injected at random times between the specified start and end time (absolute timing), but there is no provision of injecting the faults at different clock cycles. • SET can be instrumented at a random time between the specified start and end times, and an additional time specification for the hold time. Again, there is no option of mentioning specific clock cycles at which to inject the faults, or for how many cycles the hold should be exercised. The timing is absolute in nature. • Since the timing is absolute in nature, the injection times must be changed with respect to the clock cycle. Also, if the injection time is random, a lot of the faults may classify as undetected because it might not get triggered on the edges of the clock.	• Synopsys provides Timing Information that can go along with the fault type and location of the transient faults, specifying the cycles to inject faults. There is also a Timing Section in the SFF, which contains information about the cycle time, and an offset value within the cycle to inject the fault. The transient faults are valid for PORT, ARRY, FLOP and VARI location types. • Transient toggle faults take into account a start and end cycle to generate faults for an inclusive range of cycles (one fault generated for each unique cycle). Absolute timing can also be given in this case, with the desired frequency(for ex, starting from 0ns and ending at 100ns with a frequency of 10ns) to generate faults. • Synopsys provides an easy way to generate a huge space of transient faults with additional clock cycle options, and makes it easy (and extensive) to generate the transient fault space. If required, this fault space can be trimmed with the help of fault sampling. **However, it does not support transient faults on inputs, primitives and wires.**

Fault Sampling	• XFS does not have any feature to sample the faults. The fault space in a big design would typically be very large, and it is not possible to run fault simulation for the entire fault space. In order to trim the fault space in XFS, one would manually have to pick out faults randomly based on the requirements, and then carry out simulations for the same. But there is no associated confidence or coverage estimation that we can determine with such a randomized process.	• Fault sampling is provided by VCZ01X, with three supported methodologies. Statistical sampling, or confidence model sampling takes into account Confidence Interval, Confidence Level and a Coverage Estimate. Based on these parameters, a sample size can be computed and then the lower and upper bounds of actual fault coverage can be extrapolated with a given confidence. This provides an efficient and robust way of reducing the fault space, and maintaining the required fault coverage. • Percentage based fault sampling selects only a percentage of the total fault space, with the selection being random. With different seeds, the selected fault subspace will be different. Fixed number sampling randomly picks out the required number of faults from the total fault space, and changes according to the seed provided.
Strobing of signals	• The strobing of signals, i.e the observation and detection points are kept in a tcl file in a strobe list format. The strobe list can consist of functional and checker strobing points. Faults at the outputs of functional units are classified as Dangerous, and faults at the output of Safety Mechanisms are classified as Detected. With such a classification, it becomes easier to distinguish between different classes of faults as mentioned by ISO 26262, and is important to calculate different metrics. • XFS also provides an interesting strobing mechanism called virtual strobing, with the help of a system task, and allows to put virtual strobing points anywhere in the design or testbench, based on a sequence of failure events. This could be useful in complex scenarios, when faults could not be determined on the basis of faulty signals, but with the help of more complex scenarios and failure events. • There is also a checker delay window option, which allows user to keep monitoring the checker outputs for a duration of time after the functional output has a fault detection. This has a direct relation to the concept of fault tolerant time interval (FTTI) and thus, can be very useful.	• Synopsys allows to provide a list of observation and detection signals inside a FailureMode section in the SFF. Instead of the detection signals, one can put the SafetyMechanism section as well, which essentially consists of all the detection signals. There is also a separate Strobing section, if we do not want to differentiate between the two sets of observation and detection signals, similar to the single strobe list in XFS. Even though there is the provision of different classes of strobing signals, the final classification of faults does not distinguish between functional and safety mechanism outputs. A separate strobing file needs to be defined to differentiate between the two classes, and define custom fault statuses for the same. • Virtual strobing points, technically, are not a part of Synopsys Fault Simulation flow. However, it could be compared to the user-controlled fault detection tasks provided by Synopsys, `fs_drop_status` and `fs_set_status`. These could be used in the same way as the virtual strobing points, say, when a particular sequence of events are recognised or an assertion failure has occurred. • VC Z01X does not have any direct option such as the checker delay window in XFS. However, the strobing file could be set up to monitor the checker outputs for a fixed duration of time.

Fault injection	• Fault injection can be done with the help of fault ID/ fault random ID or with fault injection tcl commands generated with xfsg. This is an additional step on top of defining the fault specification file.	• VC Z01X only needs the fault target, fault type, location and timing information (for transient faults) to be defined in the SFF file. The Fault Campaign Compiler takes care of fault injection based on the FaultGenerate section defined.
Fault reporting	• XFS generates a fault report for all signals, specifying the classification, injection time, observation and detection times. There is also a final log generated containing the numbers of fault classifications.	• VC Z01X also generates a fault report, but in the format of a SFF file. This file includes details about each signal and its classification, though it lacks information regarding detection times. However, it does provide details about the test utilized for each fault, along with the corresponding Fault ID, which can be directly used to dump good and faulty waveforms.
Fault coverage and ASIL metric calculation	• XFS does not provide any in-built mechanism to generate coverage reports. The fault classifications and report needs to analyzed, followed by the manual generation of coverage reports. ASIL metrics also need to be generated manually with the fault classification numbers obtained from the final report.	• Synopsys allows the user to define custom coverage equations in the SFF. There are default equations for fault coverage and test coverage, which could be overwritten. Similarly, ASIL equations can also be approximated and defined in the SFF file to be generated automatically after running the fault simulations.

Concluding the comparison results between the two tools, VC Z01X emerges as the superior tool overall, exhibiting advanced features, precise results, and efficient fault simulation run time. It encompasses an extensive fault space with support for various location and fault types, along with features like testability analysis and fault sampling to speed up fault simulation campaigns. However, the question remains: **Is it sufficient to rely solely on this tool to devise a verification methodology that guarantees accurate fault coverage estimation for FuSa purposes?** This question is addressed in the following chapter, where we propose a methodology for FuSa verification at the RTL stage. However, before delving into this approach, we also explore the feasibility of employing formal tools to detect Safe faults within an RTL design, which could help us in optimizing the fault space and speeding up fault simulation campaigns.

4.5. Feasibility of using Formal tools

As seen in Section 3.2, identifying Safe faults is one of the first steps taken in FuSa verification to identify faults which cannot affect functional safety, and thus need not be simulated. The identification of Safe faults is aided by formal tools with their different analysis technologies as described in Section 2.3. However, we also see that FuSa EDA tools also possess capabilities for identifying Safe faults. In this section, our objective is to determine the necessity and feasibility of employing additional formal tools to identify Safe faults for RTL designs.

JasperGold Functional Safety Verification (JG FSV) app is the formal tool by Cadence used for Functional Safety. The formal flow is setup in the form of a tcl script to be invoked along with the tool, as discussed below:

- The design files are analyzed first, followed by the elaboration (along with the top module to be considered)

- Clock and reset signals are provided (for a combinational circuit, “none” option can be passed)
- Fault information is added, with the type of faults - SA0/1, SET, SEU or all, along with the fault targets, fault injection time window and SET hold time (if applicable).
- Next, the functional and checker strobes are added.
- Structural analysis is then triggered, which involves investigation of COI, activability, propagability, collapsing. Any of these options can be switched on/off to check results for a particular analysis technique. By default, all options are enabled.
- The formal properties are then generated and proofs are run. Each of these individual properties can be checked manually to see the behaviour and effect of fault. The final report is then generated with the list of Safe, Dangerous and Unknown faults.

VC Formal is the Synopsys counterpart for formal analysis, with a FuSa application mode for Functional Safety Analysis. In a way similar to JG FSV, VC Formal is also setup in a tcl script, which is then invoked with the tool. The steps for VC Formal analysis are as follows:

- The SFF files are provided to the tool in the first step with the help of which a fault database is generated internally. There is no need of providing additional fault target or strobe files, as they are taken care of in the SFF.
- In the next step, the design files are provided to the tool, and different configurations for certain type of faults need to be enabled (for example, port, primitive, array faults).
- The clock and reset information are added.
- All the different analysis techniques for Safe fault identification, i.e. structural analysis, controllability, propagation and detection analysis are carried out step by step individually to generate and check the formal properties.
- Once all the properties are generated, the final report is saved in the format of an SFF file.

The two formal analysis flows are executed on both the adder and counter designs, following the setup of strobes and fault targets as previously configured. However, initially, as there are no untestable faults in the design, none of the faults are identified as Safe by the output of the two tools. To address this, we introduce a minor modification to both designs: we connect a2 to 0 in the adder design and set counter[3] to 0 in the counter design. Following these adjustments, when the flows are rerun, SA0 faults on these two signals are identified as Safe, as expected. Further, the XFS and VC Z01X flows run on the modified designs also recognize these faults as Safe.

For the FIFO design, we execute the formal flows to determine if any additional Safe faults are identified beyond those already recognized by XFS and VC Z01X. All fault targets are enabled, covering all fault types. However, the results from the formal tool do not align with our expectations. The majority of instrumented faults are categorized as Safe, with only a limited number classified as Dangerous or Unknown, as evidenced in the results from JG FSV, shown in Figure 4.17. Even faults previously detected in fault simulation results are now marked as Safe. For example, as seen in Figure 4.18, DataIn signals are marked as Safe, when, in fact, they are not. If there is a fault on the DataIn signal, the corresponding fault will also be propagated to the DataOut signal, and is not Safe. Given that the schematic design generated does not match the level of detail provided by GLN designs, it seems that formal tools struggle to effectively utilize various analysis techniques to accurately identify Safe faults on RTL designs. **Given the limited efficiency of formal tools in identifying Safe faults at the RTL level, we opt to exclude them from the final proposed flow.** Instead, we depend on the EDA tools themselves to extract as many Safe faults as possible during their analysis of the design.

In the following chapter, we introduce a verification methodology based on the observations and results obtained from this chapter, with an aim to solve identified issues. A comprehensive outline of the verification flow is provided, accompanied by the reasoning behind the approach. The subsequent chapter presents the results obtained from its implementation.

```
---[ <FAULTS CLASSIFICATION SUMMARY> ]-----------------------
-------------------------------------------------------------
 Num  | Fault Type | Unprocessed | Safe | Dangerous | Unknown
-------------------------------------------------------------
[1]     SA0          0             293    22          0
[2]     SA1          0             254    60          1
[3]     SEU          0             60     14          0
[4]     SET          0             233    81          1
[5]     MULTI        0             0      0           0
[6]     Total        0             840    177         2
```

Figure 4.17: Formal results on FIFO from JasperGold FSV

```
---[ <SA0> ]-------------------------------------------------
-------------------------------------------------------------
 Num  | Node                                  | Classification
-------------------------------------------------------------
[1]     Reset_                                  Safe
[2]     WriteEn                                 Safe
[3]     WriteClk                                Safe
[4]     DataIn[0]                               Safe
[5]     DataIn[1]                               Safe
[6]     DataIn[2]                               Safe
[7]     DataIn[3]                               Safe
[8]     DataIn[4]                               Safe
[9]     DataIn[5]                               Safe
[10]    DataIn[6]                               Safe
[11]    DataIn[7]                               Safe
```

Figure 4.18: Formal tool classification for FIFO signals

5

Unified FuSa EDA Verification Methodology

After analyzing the tools in the preceding chapter and evaluating their results, we have gained insights into their features, advantages, and limitations. Utilizing this knowledge, we introduce a novel verification methodology combining both the tools to address the concerns arising from the individual tools.

5.1. Proposed framework for FuSa verification

The proposed verification methodology, outlined in Figure 5.1, integrates the two EDA tools discussed, and incorporates additional features and utilities. This section delves into the reasoning behind adopting such a flow, providing further elaboration and discussing the different steps and supported features within the verification process.

As demonstrated in the preceding chapter, VC Z01X offers a more comprehensive array of features compared to XFS, including fault injection capabilities across various location types and support for facilitating extensive transient fault space coverage. The primary motivation for proposing a verification flow is based on the fact that we want to address the fault space as effectively and extensively as possible, leading to accurate diagnostic coverage metrics. Despite the advanced features of VC Z01X, it remains unable to encompass the entirety of the fault space, as illustrated in Figure 5.2. While SA faults are supported on all possible location types, including inputs, outputs and intermediate nets, transient faults are only allowed on ports, variables, registers and flops, further restricting them to outputs exclusively. Consequently, transient faults cannot be injected on inputs, nets, and arrays, which may indeed occur in real-world scenarios. In contrast, XFS permits the instrumentation of SET faults on all potential locations within the design and does not limit to any location type. With this understanding at hand, we intend to develop the verification flow accordingly.

Below, we delve into the various steps comprising the methodology in detail:

- The first step in the flow is to run the stuck-at and transient fault campaigns in parallel with the help of automated VC Z01X scripts. For SA faults, we enable all possible location types and fault targets. Likewise, for transient faults, we enable all supported configurations. With VC Z01X, there are two options for enabling transient faults: transient toggle and transient hold. Transient toggle injects the fault at a specific clock cycle and maintains it until the subsequent cycle, after which the value changes based on the input. With transient hold faults, a fault can be injected at a specific clock cycle and retained until the designated cycle (Equation 4.3). Transient hold can be seen as an extension of transient toggle faults. The decision to utilize these two fault types is made in the SFF file through the corresponding statements.

  In the default proposed flow, we focus on exercising transient toggle faults and specify start and end cycles for fault injection, as also shown in Equation 4.2. Faults are injected individually at each cycle within the given range. The choice of these two points (start and end cycle) depends on the toggle activity of the design and the possibility of faults being detected at the strobes. For example, in the FIFO design, writes start happening to the FIFO from the 3^{rd} clock cycle and end at 12^{th}. Subsequently, reads start happening from 12^{th} cycle onwards. Thus, if we provide

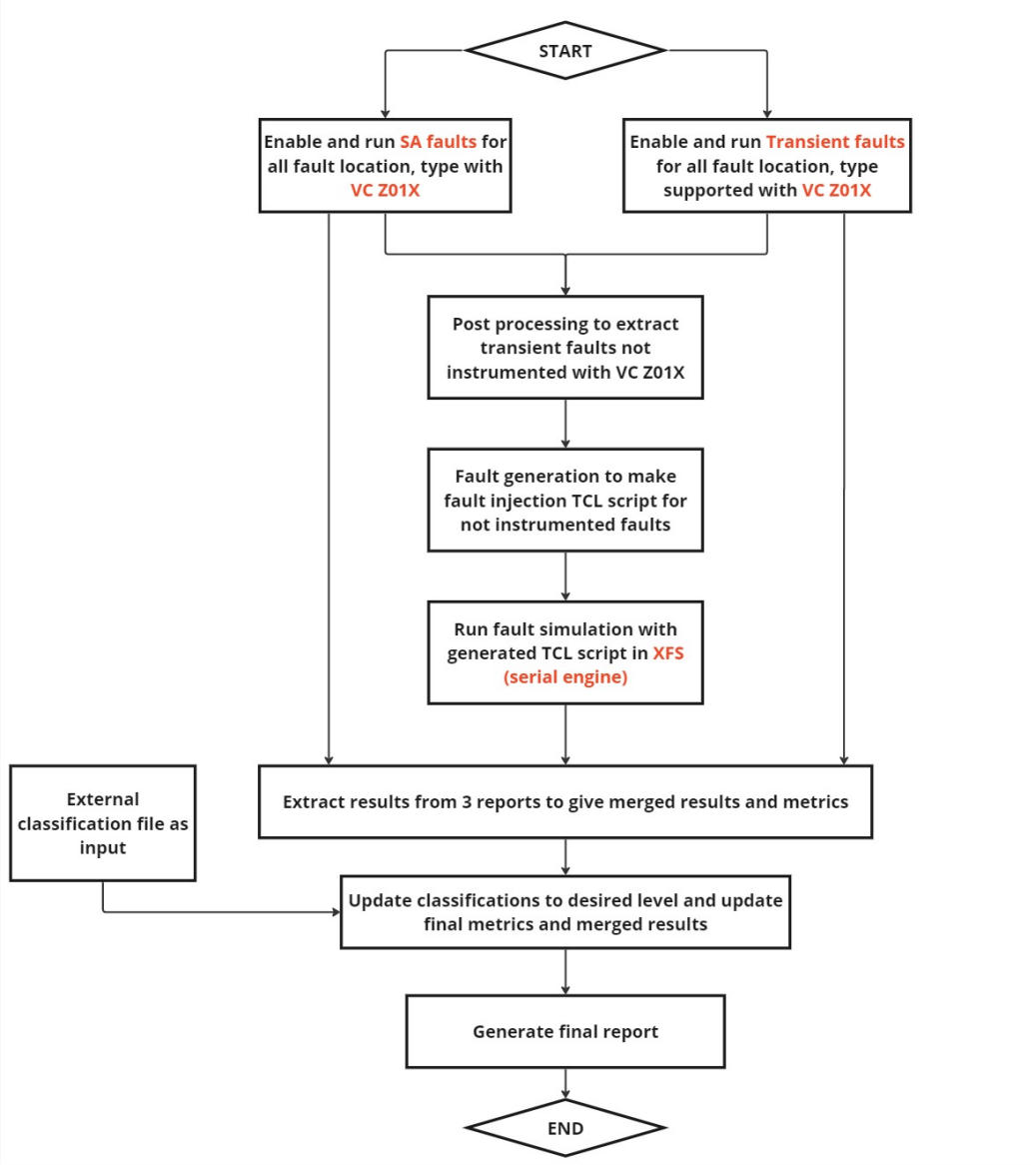


Figure 5.1: Proposed verification methodology using combination of EDA tools

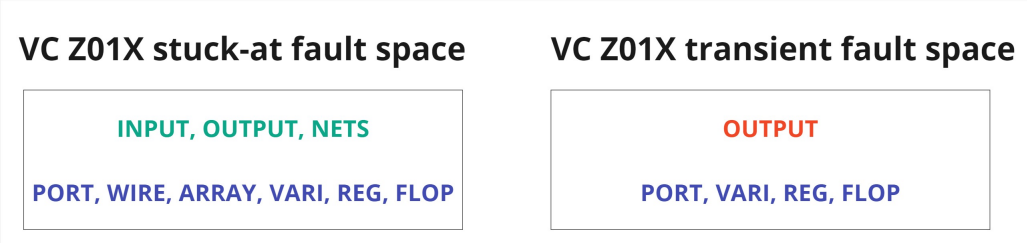


Figure 5.2: Fault space covered by VC Z01X

the start cycle as 3, and end cycle as 15 (or any number greater than 13), we might be able to see the effect of faults on the functional and checker strobes. Based on such information, we provide the start and end cycles in the SFF file for transient faults. In summary, we run SA and transient fault campaigns using VC Z01X on all supported locations and types to obtain the initial results.

- We know that transient faults cannot be applied to specific locations like nets, inputs, and arrays.

Consequently, the reports generated from the first step will highlight discrepancies in signals not targeted by the VC Z01X flow. This indicates signals that have not been subjected to transient fault injection using VC Z01X. Therefore, the next step involves extracting information from these reports to identify all such signals. This process utilizes regular expressions to iterate through each signal in the SA list and verify its presence in the transient report. If no match is identified, it indicates that the fault location has not been subjected to transient fault injection with VC Z01X. Now, these signals can be injected with SET faults using XFS, as this tool doesn't impose any limitations on the placement of SET faults.

- The next step is to write fault injection commands for these signals in a tcl format, which can then be invoked with XFS. To achieve this, we expand upon the logic employed in the preceding step for generating fault injection commands. We consider various parameters such as clock period, reset time, start cycle, end cycle, and the range for hold time start and end. Listing 5.1 shows a portion of the logic employed to generate the relevant commands.

```
1    if fault_type == "0" and location_type != "PORT":
2        for i in range(start_cycle,end_cycle):
3          inject_time = reset_time + i*clock_period
4          set_hold_time = random.randint(hold_time_range_start,hold_time_range_end
     )
5          #Write fault injection command in output tcl file
6          output_file1.write("fault -inject -time %dns -type SET+%dns {%s};run;
     reset\n" %(inject_time,set_hold_time,signal))
```

Listing 5.1: Generation of Fault injection command

Each fault recognized as not injected by VC Z01X is processed by the code displayed. To prevent generation of duplicate commands, we ensure that we only search for SA0 fault type for the specific signal. This restriction is applied because any signal not included in the transient campaign will appear in the SA report for both SA0 and SA1, resulting in redundant fault injection commands if this constraint is not enforced. Moreover, considering that VC Z01X injects faults on different location types, implying that port faults are also instrumented as wire faults, we opt for only one of these two faults. This process is also repeated for the collapsed signals identified in the VC reports which have not been injected with transient campaign.

Since fault injection in VC Z01X is made to start after reset has been asserted, the injection time for XFS is accordingly adjusted, in line with the fault modeling done for transient faults in Figure 4.14. The **start and end cycle** parameters correspond to those used in the transient fault generation statements of VC Z01X. As we are injecting SET faults for the identified signals, we also need to specify a hold time. This can be set to a default value of the clock period. However, to introduce variability in the transient faults being injected, we include a **hold time range**. A number from this range is randomly generated and added as the hold time to the fault injection command. The process iterates over each signal and the complete range of faults to generate a fault injection TCL file for use in the subsequent step.

- The XFS script earlier developed is now invoked as: `./xfs_flow.sh serial custom trans`. Using these command line arguments, all transient faults are instrumented, but only the SET faults present in the custom TCL file generated in the previous step are executed. Since, SET faults are not supported with the concurrent engine, we have no choice other than to invoke the serial flow. If necessary, multiple licenses of XFS can be used to divide the fault injection commands and execute them simultaneously to accelerate the process.

- Following the execution of the XFS flow for transient faults, we consolidate the three reports — those from VC Z01X for SA and transient faults, and the XFS report — utilising a Python script that employs a variety of regex operations. These results are merged and compiled into a final report, presented in the form of an SFF file with some necessary modifications. Initially, we extract the individual test results from the three reports to provide insights into the numerical data associated with each flow. This contains the fault classification counts, excluding the Safe/Untestable faults.

  The classifications from XFS are mapped to their corresponding VC Z01X classifications to ensure consistency in naming across the results and to eliminate any confusion arising from the different

classification names used by both tools. Due to the fewer classification categories in XFS compared to VC Z01X, it is logical to adjust the XFS fault classes to align with the naming conventions used in VC Z01X. For each report, a classification array is introduced to store the fault classification numbers. For instance, if the total number of fault classes was 30, as specified in VC Z01X, three arrays of size 30 were generated. An enumeration is implemented to represent the various fault classes and assign numbers from 0 to 29 accordingly. These enumeration numbers could then be utilized to access the arrays corresponding to the fault classification. If there's a need to add or remove fault classes, only the enumeration requires modification, ensuring the script's continued functionality

The classification numbers are extracted from each report and appended to the corresponding index in the array specific to that report. Once all numbers have been incorporated into their respective arrays, they are aggregated to obtain the classification numbers for the final report. Since VC Z01X provides numbers for various fault groups (consisting of a group of fault classes), it is important to calculate the counts for these groups too. Finally, Diagnostic Coverage is computed based on the consolidated classification numbers.

The output report generated at this stage consists of the following information:

1. Test results from the three reports (excluding Safe faults)
2. Information regarding Fault statuses (classes), Promotion Table and Coverage equation (Diagnostic Coverage)
3. List of signals with their respective classifications (from 3 different reports)
4. Fault classification numbers for 3 individual reports, including their Diagnostic Coverage metrics.
5. Merged fault classification numbers and updated Diagnostic Coverage.

- The final step of the process addresses a notable absence in the tools. Occasionally, we possess knowledge indicating certain signals are not dangerous and can thus be upgraded to Safe status. Additionally, there are instances where we anticipate certain faults will always be detected at functional outputs and remain undetected by Safety Mechanisms, based on prior designer input. However, the tools lack a provision for manually adjusting the fault status of these signals to the desired classification. To address this limitation, we introduce a feature in this flow wherein a classification file is provided as input, specifying the signals slated for status conversion along with the source and destination classifications. This enables the adjustment of fault classifications for the specified signals and facilitates the updating of the final report, including the Diagnostic Coverage metric.

  An example of the input file which goes into this step is shown in Listing 5.2. The list includes signals alongside their respective source and destination classifications, with an additional parameter introduced to offer user customization. This parameter takes one of two options: "group" and "single", allowing users to specify whether they wish to update the classification for the single signal only or for all collapsed signals associated with it as well.

```
1    test.DUT.sdpram_i1.ECC_Enc.DataEnc[1]              IX UU group
2    test.DUT.sdpram_i1.ECC_Enc.DataEnc[4]              ND NN group
3    test.DUT.sdpram_i1.sdpram_i1.L_DataIn[6]           IX UU group
4    test.DUT.DataIn[0]                                 ON OD single
5    test.DUT.sdpram_i1.ECC_Dec.DataOut[0]              ON UU single
6
```

Listing 5.2: External input file for manual classification

To update the classifications, each line in the input file is parsed to extract the signal and the intended classification for conversion. Subsequently, the report generated in the preceding step is searched to find the specified signal. If the source classification matches, the fault class for that particular signal is then updated to the destination classification. In addition to this, it is also important to update the classification numbers for the individual reports as well as the final reports. This involves considering the third parameter (group or single) to determine whether to update a single signal or a group of collapsed signals. For a group classification, all collapsed

signals must be converted to the desired fault class, and the fault class number for both the source and destination classifications needs to be adjusted based on the number of signals in the collapsed list. The numbers in the individual arrays are revised and then merged to form the final classification array. Furthermore, the individual Diagnostic Coverage metrics of the reports are updated, along with the computation of the final metric. The original report and the modified report are kept as two separate files in order to see the differences before and after the final step of the flow.

In addition to this, the workflow includes the capability to independently conduct fault simulations with XFS when deemed necessary, facilitated by the XFS script. Comparison scripts are also maintained to enable the execution and subsequent comparison of the results of both tools, thereby providing discrepancies in classification. This allows the user to run flows with either of the tools individually based on requirements and analysis of results.

This marks the conclusion of the outlined verification methodology aimed at conducting Functional Safety Verification on RTL designs. In the following chapter, we delve into the outcomes of applying this methodology to the previously discussed FIFO design, comparing the resulting diagnostic coverage from individual tools with that of the proposed verification flow. Additionally, we examine the effects of the classification update feature incorporated into the verification flow. Furthermore, we apply this methodology to an automotive System-on-Chip (SoC) design, characteristic of those commonly found in automotive chips, and analyse the results to identify areas for enhancement.

6

Results

In this chapter, our objective is to apply the verification flow to the FIFO design and an automotive SoC. We compare the results and Diagnostic Coverage obtained from this approach with those achieved by running the two tools separately. Further, we also look to update the classifications of signals using the developed feature and verify the results. Since the FIFO design represents a small-scale study of Functional Safety (FuSa) verification, we also extend our analysis to a larger, more intricate SoC design. In this design, we look to identify areas of improvement after running the verification flow, and make enhancements for the same. This is further elucidated in the following sections.

6.1. Classification results of flow on FIFO design

Table 6.1 lists the results of SA and transient faults injected across all supported locations in the FIFO design, combining the results from the SA and transient fault analyses conducted with VC Z01X, as highlighted in Section 4.4.2. Faults classified as NC, UU, and NN are deemed Safe and therefore excluded from the Diagnostic Coverage calculation. IX faults are categorized as Dangerous because their effect on the functional and checker outputs remains unknown, as VC Z01X labels faults inserted at locations in an 'unknown (x)' state as IX. Hence, these faults are factored into the denominator of the DC equation. Manual debugging is required to assess whether these faults pose a danger or are Safe in practical scenarios, and if any modifications to the design could mitigate the occurrence of an unknown state. Utilizing the available data, the DC of the FIFO design with VC Z01X is computed to be 80.47%.

Table 6.1: Summary of all faults instrumented on FIFO with VC Z01X

Classification	**VC Z01X**
Not Observed Diagnosed (ND)	876
Not Observed Not Diagnosed (NN)	2396
Observed Diagnosed (OD)	579
Observed Not Diagnosed (ON)	674
Total	**4525**
Impossible x-state (IX)	48
Untestable Unused (UU)	616
Not Controlled (NC)	223
Total	**5412**
Diagnostic Coverage	**80.47%**

The SA fault results generated by XFS are already compiled in Table 4.6, with a total of 706 faults identified by the tool. Similarly, transient fault results are also documented, as depicted in Table 4.7. However, this does not take all faults into account, as the fault instrumentation was formulated utilizing a report from VC Z01X. Consequently, we execute the XFS workflow independently for transient faults, providing distinct cycles for injecting SET and SEU faults (ranging from the 3^{rd} to the 13^{th} cycle, same

as VC Z01X), with adjustments made to the script accordingly. The transient results solely from XFS, alongside the total results combining both SA and transient fault campaigns, are presented in Table 6.2.

Table 6.2: Summary of all faults instrumented on FIFO with XFS

Classification	XFS transient	XFS overall
Not Observed Diagnosed (ND)	2200	2432
Not Observed Not Diagnosed (NN)	1184	1370
Observed Diagnosed (OD)	421	577
Observed Not Diagnosed (ON)	1074	1162
Total	**4879**	**5541**
Impossible x-state (Injection failed) (IX)	192	192
Untestable Unused (Untestable) (UU)	20	64
Total	**5091**	**5797**
Diagnostic Coverage		**68.96%**

The IX fault status is defined as “Not Injected” in XFS, while the UU fault status is labeled as “Untestable”. As seen from the two flows, there are differences in the final classification metrics and the diagnostic coverage numbers. First, the higher count of SA faults in VC Z01X is attributed to the wider range of supported location types. In contrast, XFS instruments more transient faults as there are no restrictions on the placement of such faults at inputs and intermediate nets. Additionally, with transient faults injected at varying timestamps, the overall count of XFS faults exceeds that of VC Z01X. Diagnostic coverage numbers also vary between the two tools, primarily due to differences in classification numbers. The DC for XFS is on the lower side, owing to the large portion of transient faults at inputs being observed at functional strobes and not detected at checker strobes. Moreover, the presence of multiple injection times results in a manifold consideration of faults, potentially reducing DC if the same classification recurs. This also highlights a challenge with transient fault campaigns - the expansive fault space and the lack of standardized guidelines for assessing transient fault space and injecting faults. Despite this, we proceed with treating each fault injected at a specific time as distinct, as faults occurring at different time instants may exhibit varying behavior and thus warrant separate consideration.

Table 6.3: Summary of all faults instrumented on FIFO with the proposed verification flow

Classification	Fault numbers
Not Observed Diagnosed (ND)	2611
Not Observed Not Diagnosed (NN)	1393
Observed Diagnosed (OD)	731
Observed Not Diagnosed (ON)	1252
Total	**5987**
Impossible x-state (IX)	156
Untestable Unused (UU)	616
Not Controlled (NC)	223
Total	**6982**
Diagnostic Coverage	**70.36%**

The results of the proposed verification methodology applied to the FIFO design are depicted in Table 6.3. This analysis does not include an external classification input file to automatically modify the outputs of specific signals, which will be discussed in the next step. As evidenced by the results, the proposed flow yields a DC of 70.36% for the FIFO. This DC considers all possible fault locations in the design for SA faults. Transient faults on supported location types with VC Z01X are considered, following which the remaining fault space is covered by XFS. It is however interesting to note that the back propagation issue would still be prevalent for input ports while instrumenting transient faults with XFS. Although it represents a real life scenario and signifies a fault injected on the wire connected to the port, the transient fault space in XFS does not cover faults injected at the port inwards towards

lower hierarchies. Consequently, a small subset of faults remains unaddressed by the overall verification flow, a shortcoming which could not be solved by the combination of the tools as well.

Following this report, the next logical step involves identifying methods to enhance coverage. To achieve this, all dangerous faults must be analysed. For instance, ON faults identified during SA fault analysis should be rectified prior to injecting transient faults. Otherwise, the percentage of ON faults may significantly increase during transient fault injection with the addition of different time instants. Similarly, IX faults require examination to determine if they can be reclassified or if design alterations can be made to assign appropriate values to the signals, thereby eliminating the unknown state.

To verify the added functionality of updating classifications, we consider the input classification file, as mentioned in Listing 5.2. We present the results of the reports, before and after integrating the classification file as an input to the flow. Considering the signal `test.DUT.sdpram_i1.ECC_Enc.DataEnc[1]`, where we want to convert all IX classifications of the signal to UU, we see the following results, as shown in Figure 6.1 and Figure 6.2. **The original report is displayed on the left side, while the modified report is shown on the right side.** In Figure 6.1, the signals are found, but they do not match the source classification. Hence they are not updated in the final report. In Figure 6.2, all signals have matching source classifications, resulting in their update to UU classification. Further, the number of IX classifications decreases by 4, while the corresponding UU class increases by 4.

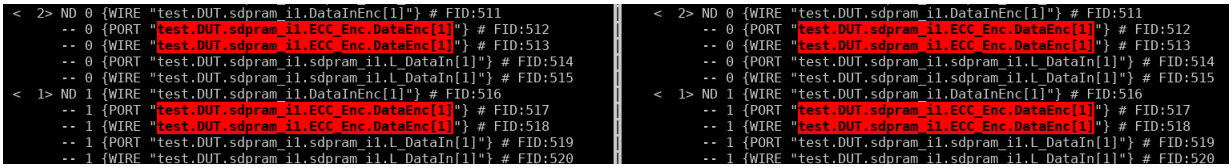


Figure 6.1: Conversion of signal using classification script - no match found

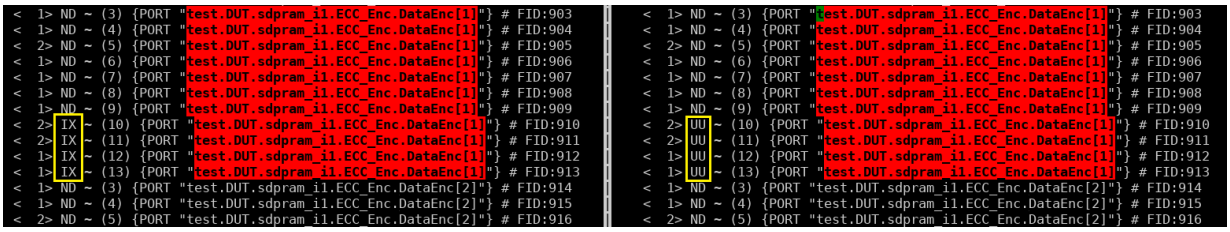


Figure 6.2: Conversion of signal using classification script - match found and updated

Now, considering the signal `test.DUT.sdpram_i1.ECC_Enc.DataEnc[4]`, depicted in Figure 6.3, we observe that this signal is part of a collapsed fault group involving multiple signals. Since the 'group' parameter is specified, all classifications in the list need to be updated. Consequently, the primary fault in this list is adjusted to NN, the desired classification. The collapsed signals, denoted by '- -', remain unchanged as they inherit the fault class of the primary fault. In terms of numerical changes, ND decreases by 10, while NN increases by 10.

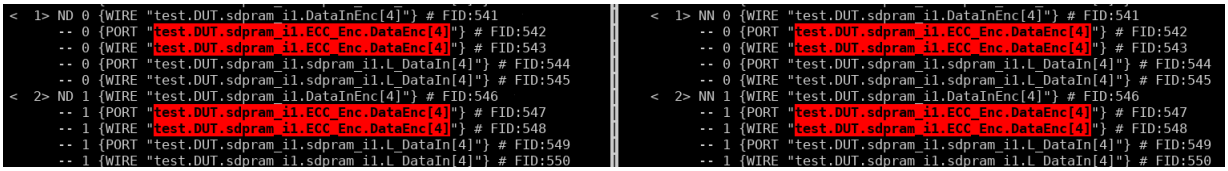


Figure 6.3: Conversion of collapsed signal in a list (with parameter 'group')

Now, let us consider the case of `test.DUT.DataIn[0]`, which requires conversion from ON to OD. However, the 'single' parameter is also applied to this signal. This implies that if this signal is part of a collapsed list, the classification of all other signals in the list should remain unchanged. The corresponding reports are depicted in Figure 6.4. The required signal is changed to the desired classification, whereas the other two signals in the list are kept as ON. Further, OD is incremented by 2, while reducing ON by 2.

Figure 6.4: Conversion of collapsed signal in a list (with parameter 'single')

Considering the external classification file(Listing 5.2), we see that the Diagnostic Coverage increases from 70.36% to 70.81%. However, the input file considered in this case was developed in order to test different scenarios for the verification of the added feature, and not representative of actual modifications required for signals. Therefore, we look at a more practical case of the IX faults to see if they can be changed to a more accurate classification based on manual analysis of the faults, and utilise this feature.

To recap the verification setup for FIFO design, writes were initiated to all locations of the FIFO until 1250ns, after which reads were triggered. Taking a look at the 176 IX faults present in the design, there are several groups of signals which contribute to this - first, `test.DUT.sdpram_i1.ECC_Enc.DataEnc` and `test.DUT.sdpram_i1.sdpram_i1.L_ DataIn`. All bits of these signals are currently classified as IX when the corresponding bit is held at an x-state. Upon closer observation, **it appears that no value is being driven to these signals in the testbench after the writes have been completed, resulting in them being held in an unknown state.** However, these signals are detected by the Safety Mechanisms at all other timestamps. In this scenario, we have two options: first, we can modify the testbench to accurately drive the appropriate signals, or second, we can utilize the classification update feature to adjust the fault classes of these signals. Given the availability of the feature, we opt to utilize it to convert these IX faults.

Next, we have the set of signals - `test.DUT.DataIn`, `test.DUT.sdpram_i1.L_DataIn`, `test.DUT.sdpram_1.ECC_enc.DataIn`, which are Observed and Not Diagnosed (ON), until the writes are being performed. Once the read operation starts, these signals are again held at an x-state. However, if these signals were set to a particular value(ideally 0 if no write is happening), and a transient fault was injected on them for a clock cycle, they would neither be observed nor diagnosed (NN) as it would not affect the read functionality happening at that time. Hence all these IX faults can be converted to NN.

After converting the aforementioned set of signals to the required classifications, we are able to improve the Diagnostic Coverage from 70.36% to 73.23%. An important thing to consider is that there are still 223 'Not Controllable' (NC) faults, as seen in Table 6.3. These faults are not representative of any group of signals which can be updated to a required classification based on manual analysis. Therefore, tests must be written in order to toggle the signals at these faults. Consequently, these faults could fall into the OD/ON/NN categories based on the test outcomes, thus impacting the DC once again. As the initial two tests conducted for this design were targeted, a third test is planned to introduce some randomness in the data written to the FIFO, aiming to cover a broader range of data combinations.

The number of NC faults decreases from 223 to 58 after executing the flow with the third test. These faults drop into the categories of NN, ON, and OD. Consequently, the count of ON faults increases from 1252 to 1304, **resulting in a slight decrease in DC from 73.23% to 73.21%**. This indicates that by introducing more randomness and toggling additional signals in the simulation, the DC experiences a marginal reduction. While further tests could be developed to address the toggling of signals associated with NC faults, this aspect is not pursued here, considering this as merely an illustrative example for FuSa verification. However, in a more comprehensive design, such considerations would be crucial, prompting the development of more refined tests.

We conclude the takeaways from the results of the proposed verification flow in the following points:

- The Diagnostic Coverage achieved through the verification methodology takes into account the maximum possible fault space supported by a combination of the two tools, resulting in a more accurate coverage estimation. However, the transient fault space analysis is still limited due to the lack of standardized guidelines and information regarding the choice of fault injection and hold times.

- The faults classified as ON, OD, NN and ND can be taken on merit from the results of the final report. Other signals such as IX and NC need to be deduced in order to further classify the signals.

- Transient faults categorized as IX need to undergo analysis to determine whether the design can be enhanced to prevent those signals from being held in an unknown state at any instance after the reset has been deasserted. Alternatively, if they can be assumed safe on a detailed analysis of the faults at other timestamps, they can be converted to the desired classification using the proposed classification update feature.
- Faults labeled as NC indicate signals that do not toggle. Therefore, more effective tests need to be developed to enable these signals to toggle during good simulations. Additionally, if these signals collectively represent a group that is typically detected or considered safe, they can be reassigned to a desired classification.

The analysis of the verification methodology on the FIFO design provides valuable insights into the effectiveness of Safety Mechanisms and serves as a preliminary step in Functional Safety Verification. However, it is important to note that this design is not fully representative of automotive industry standards. Therefore, we shift our focus to a more relevant automotive design to analyse the results and focus on enhancing the design's reliability and robustness.

6.2. AutoSoC overview

To assess the efficacy of the verification flow on a larger scale, it is important to select an appropriate benchmark design tailored for automotive needs. In view of this, we opt for the Automotive SoC (AutoSoC) [51] [52], a fully open-source benchmark suite for automotive System-on-Chip (SoC) that encompasses both hardware and software components. The AutoSoC was created with the intention of providing a benchmark suite capable of meeting industry demands while also being open to researchers for conducting comprehensive comparisons across different methodologies and tools. Hence, it serves as an ideal candidate for us to execute the proposed verification flow and analyze the results.

The AutoSoC is distributed as a collection of configurable hardware IPs, described in RTL, and integrated into a System on Chip. The suite comprises various hardware configurations, diverse options of Safety Mechanisms (SMs), and software applications developed for the automotive sector. Figure 6.5 presents an overview of the functional blocks in AutoSoC. The concept of functional blocks is introduced to maintain modularity within the design. By tailoring different versions of AutoSoC to meet the specific requirements of each functional block and accommodate various design use cases, the flexibility to incorporate different hardware components is ensured.

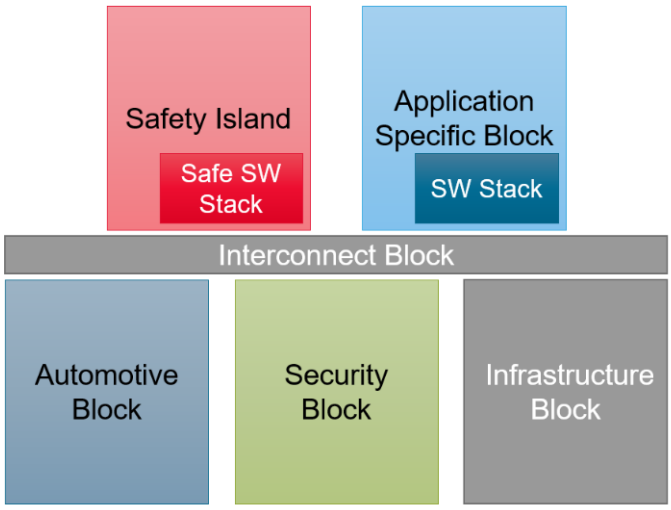


Figure 6.5: Functional modules of AutoSoC [51]

The Safety Island serves as a dedicated functional block responsible for handling all safety-critical processing tasks. It encompasses essential components such as the CPU and memories, which require protection through Safety Mechanisms. On the other hand, the Application Specific Block consists of hardware tailored for application-specific processing needs. This unit may incorporate CPUs and

memories optimized for high-demand applications, as well as specialized components like GPUs and image processing units tailored for video applications.

Complementing these primary blocks are auxiliary components essential for broader system functionality. The *Automotive Block* facilitates communication with in-vehicle systems, supporting CAN protocol. However, protocols such as FlexRay, LIN, and Automotive Ethernet could also be incorporated in this block. The *Security Block* manages all security-related functionalities, encompassing encryption cores such as AES and DES. The *Infrastructure Block* oversees online health monitoring, equipped with debugging tools like JTAG and UARTs to ease development processes. Lastly, the *Interconnect Block* governs internal communication, utilizing standard buses such as AXI and Wishbone or advanced solutions like Network-on-Chip (NoC) to facilitate easy data exchange between components.

The CPU used for AutoSoC is based on the mor1kx implementation of the OpenRISC CPU [53]. The SoC is based on an example SoC package comprising of a CPU, memory, UART, JTAG, and a debug unit, interconnected through a Wishbone bus. Additionally, the SoC incorporates a testbench equipped with functionalities for loading software applications into memory and establishing a connection to the debug unit via JTAG. On the software side, the AutoSoC integrates different software resources to cater to different application scenarios. This includes an extensive set of development tools, pre-compiled test applications, a bootable Linux kernel, and a dedicated RTEMS[54] environment.An Automotive Cruise Control application is available (`autosoc.exe`), built upon the RTEMS operating system, which includes real-time tasks for vehicle sensor data processing, actuation computation, engine parameter adjustment, and system housekeeping. This is also the software application which is used primarily while executing the verification flow on the AutoSoC.

Our main emphasis lies on the inclusion of Safety Mechanisms within the Safety Island, a vital component for safety-critical applications. AutoSoC offers support for various Safety Mechanisms (SMs), each customizable through additional defines provided during elaboration and runtime. Using these defines, different configurations of AutoSoC are defined, pertaining to different ASILs. Figure 6.6 presents one such configuration (SAFE configuration), which implements all the Safety Mechanisms - DCLS, ECC, Bus Parity, Checkpoint Control, Safety Monitor and STLs. The different SMs are discussed below:

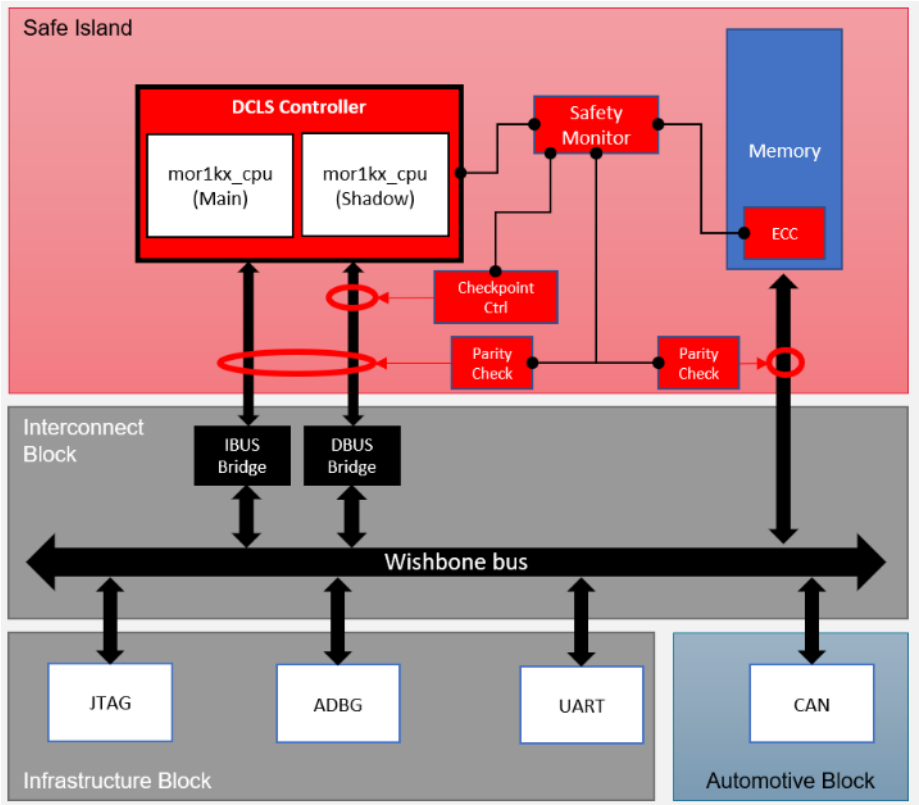


Figure 6.6: AutoSoC SAFE configuration [51]

- **Dual Core Lock Step (DCLS)**: DCLS is the most common SM which is present in processors targeting ASIL D applications. A time diverse DCLS is implemented in AutoSoC as one of the configurations, as shown in Figure 6.7. Only the main CPU has write access to the bus, controlling the SoC's functionality, while the shadow CPU's outputs are used for fault detection by the Compare Unit. Any discrepancies between the processors' outputs trigger an alarm through the Compare Unit. The delay units are inserted at the output of the main CPU and the input of the shadow CPU to provide temporal diversity. The default configuration implements a delay of 2 clock cycles, which can be changed according to the requirements of fault tolerant time interval.

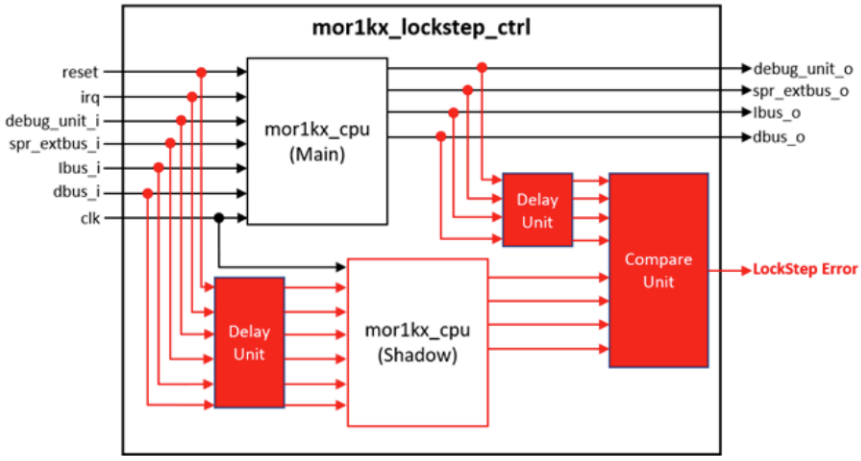


Figure 6.7: DCLS implementation in AutoSoC with time diversity

- **Internal and External Memories ECC**: Internal memories typically occupy a large area on the physical device, thereby correlating to the maximum probability of faults. Based on industry standards, ECC emerges as the best option for implementing SMs for internal memories, considering it's high probability of detecting and correcting faults while maintaining a relatively low area footprint. Additionally, external RAM should also be protected by ECC since software applications depend on external memory for storing critical data and control parameters. Consequently, failures in external memory could directly impact intended functionality, thereby requiring protection by ECC.

- **Bus parity**: The implementation of a parity checker to the data bus ensures fault detection in data transmissions between the CPU and memory, protecting against faults that may otherwise go undetected by other SMs. The parity checker calculates a Parity bit for each communication happening between the memory and CPU, and any discrepancy triggers an alarm to alert the system.

- **Checkpoint Control**: The Checkpoint Control monitors the Data Bus for predetermined software signatures in specific memory locations, protecting against common mode failures that could happen in DCLS. Acting as a Hardware Watchdog, it verifies the Control Flow by expecting different signatures for each software task, ensuring not only that the software is running but also that the Control Flow aligns with expectations. This configuration is cutomizable in terms of software signatures which can be provided along with the expected sequence and deadlines.

- **Safety monitor**: The Safety Monitor block generates external alarms and error codes upon detecting faults in any Safety Mechanism, serving as a central hub for detection alarms. As shown in Figure 6.6, this block considers signals from DCLS, External Memory ECC, Bus Parity, and Checkpoint Control, to generate error alarm and provide the specific code for the triggering SM.

- **Software Test Library (STL)**: STLs are integrated in AutoSoC to minimize overhead in terms of area while still achieving high ASIL capability. The STL consists of 16 test programs totaling 64

KB. The AutoSoC STL Configuration focuses on the CPU (mor1kx_cpu), eliminating potential sources of non-determinism like Instruction Cache and Data Cache. It is important to ensure a deterministic instruction stream for STL execution, including deactivation of modules causing control flow fluctuations (e.g., caches) during fault grading. This setup does not prohibit the use of caches or similar components during STL deployment but requires additional preparation for test library execution.

6.2.1. AutoSoC setup

The AutoSoC benchmark suite supports different configurations, combining a choice of Safety Mechanisms to be used, along with the software application to be tested. On the hardware side, the different configurations[51] supported by AutoSoC are outlined in Table 6.4.

Table 6.4: AutoSoC configurations [51]

Benchmark configuration	DCLS	Internal Mem ECC	STL	Bus parity	Checkpoint Control	Safety Monitor
QM	-	-	-	-	-	-
ECC	-	+	-	-	-	-
STL	-	+	+	-	-	-
DCLS	+	-	-	-	-	+
SAFE	+	-	-	+	+	+

In order to create a new configuration, a separate build directory needs to be created containing the following files:

- An **elaboration file** containing all the design sources and testbench files must be provided. Unless new design or TB sources are added, this file remains consistent across all configurations.
- An **additional args file** is to be provided containing the different configurations to be run with AutoSoC. For example, the software application to be run must be provided in this file along with `+elf_load=<software binary>` option. In order to run the autosoc application, we provide the option - `+elf_load=autosoc.exe`. With this option, the binary file is loaded onto the memory for simulation. `clean_ram` is used to write zeroes to the RAM memories during the start of the simulation.

  In order to configure the Safety Mechanisms, additional defines need to be provided. For example, if we want to enable ECC on internal memories, bus parity and checkpoint control in one configuration, we need to provide the following options:

  - *+define+MEMECC*
  - *+define+PARITY*
  - *+define+CHECKPOINT*

  For DCLS, the define to be used is *+define+LOCKSTEP*. Although Safety Monitor is provided as an option which can be configured, the AutoSoC benchmark implements it regardless of any additional defines. Therefore, this monitor is enabled by default, requiring no additional defines. Likewise, for the STL configuration, there's no need for extra defines to be provided, except for altering the software application to be loaded. The STL application needs to be loaded for the STL configuration to work.

- **Functional and checker strobes** must be specified for the given configuration. The functional strobes remain consistent across all configurations and include signals related to the instruction and data buses, as well as Special Purpose Register (SPR) accesses to external units like cache and MMU. Regarding checker strobes, the relevant signals should be included depending on the configuration used. For instance, if ECC is implemented, the corresponding error detection signals from ECC decoder modules must be designated as the checker strobes.

- The **fault targets and exclusions** are dependent on the user and configuration used. They can be adjusted according to specific requirements and the necessity of module-level fault injection.

Using the described setup, automated scripts facilitate the generation of fault injection simulation results on the AutoSoC. In the following section, we look at the initial results and the implemented enhancements designed to elevate the ASIL level of the SoC.

6.3. Implementation of AutoSoC Safety Mechanisms and classification results

First and foremost, it is important not to proceed directly with fault simulation on the entire design because of it's large and complex nature. Instead, dividing the design into smaller blocks aids in facilitating easier analysis. By examining the total number of instrumented faults across different parts of the design, we gain insights into which areas cover the most fault locations. As also seen in [51], faults are injected on all possible locations in the `mor1kx_cpu`, including the internal memories. The initial analysis involves obtaining the list of SA faults generated by XFS, utilizing the existing setup in the AutoSoC benchmark. As depicted in Figure 6.8, internal memories encompass approximately 96% of the fault space, with logic faults in the CPU constituting the remaining 4%. Further distribution of memory faults is illustrated in Figure 6.8 as well. Given that these memory faults dominate most of the fault space, prioritizing them seems logical. Addressing these faults initially could potentially elevate the component to an ASIL B level, even if the ECC can detect 99% of the faults.

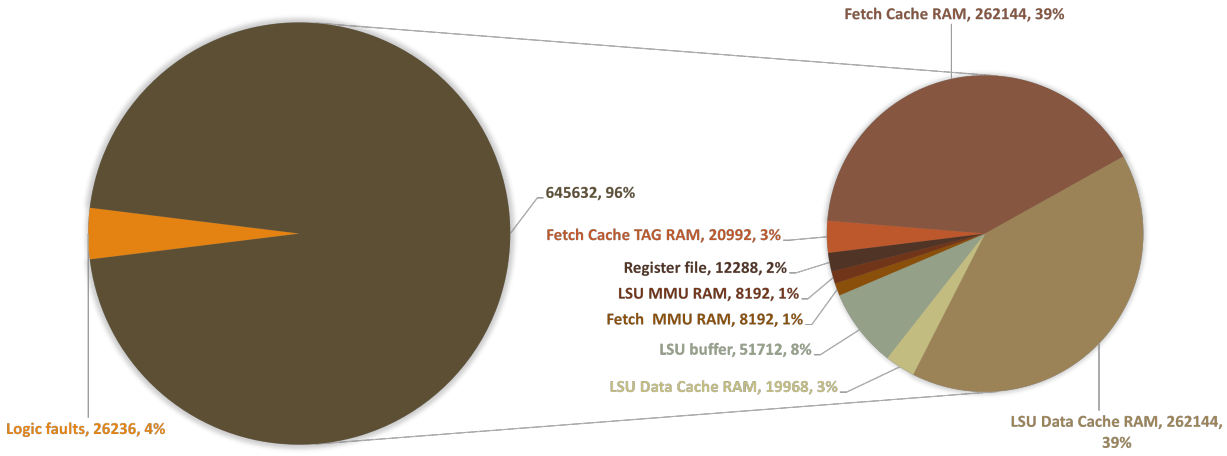


Figure 6.8: Distribution of logic and memory faults

[51] does not consider the register file memory as part of the internal memories, but instead considers it a part of the logic faults. Hence, the corresponding analysis presents that internal memories only cover 91.3% of the fault space. The Diagnostic Coverage of ECC is estimated as 92.8%, considering a 99% DC, in [51]. However, it is important to note that fault simulations have not been performed in this instance, and the figure serves as an estimate of the DC. To validate this estimate, we perform fault simulations on the design.

At first glance, after running fault simulations on one of the internal memories (Fetch Cache Tag RAM), **we see that none of the faults on the memories are getting detected, and hence the Diagnostic Coverage obtained from the ECC Safety Mechanism is 0%.** Upon closer observation, it is noticed that ECC is not implemented for the internal memories of the AutoSoC. Therefore, utilizing the available ECC modules in the benchmark suite, ECC is implemented on internal memories, which is further discussed in the next section.

6.3.1. Results of ECC implementation on Internal Memories

All internal memories are instances of dual-port RAM (DPRAM) module. To enable ECC on memory, a separate DPRAM module with ECC functionality is developed. An OPTION_MEMECC parameter is passed to various modules in the hierarchy based on whether lower modules require an instance of the DPRAM. Depending on the status of OPTION_MEMECC, either a generic instance or the ECC-enabled instance of the DPRAM is invoked.

The ECC module available in AutoSoC is a double error correcting (DEC) BCH [55] code. It allows for correction upto 2 bit errors, and supports 16/32/64/128 bit memories. It operates on complete memory words in a single cycle and is a purely combination logic design. When data is written to a particular address in the RAM, the same data is also passed as an input to the ECC encoder module. As depicted in Figure 6.9, the ECC encoder module takes a data input (d_i) and computes an ECC syndrome (p_o) (calculated ECC signature) based on the data width. For instance, with a data width of 64 (upto 113), the corresponding syndrome length is 14. Similarly, for a data width of 32 (upto 51), the syndrome width is 12. This parametrization allows for flexibility regarding the supported data widths. For example, most internal memories have a data width of 32 bits. However, certain instances of the DPRAM memories in modules like Tag RAM and LSU store buffer have data widths of 39, 41, and 102 bits. Depending on the data width, the syndrome width is adjusted accordingly, thereby also requiring careful specification of signals of that particular width connected to the port. The calculated ECC signature is specific to a particular address and is used for comparison while decoding the read data from this address.

The decoder module operates with two inputs: the read data from the address (d_i) and the stored ECC signature at that address (ecc_i). Using these inputs, the module sets the corresponding error detection signal (err_det_o) and mask (msk_o). If no error is detected, the mask remains 0. However, if an error is detected, the mask becomes a non-zero value, indicating that the read data must be XOR-ed with the mask before being assigned to the data out signal from the RAM.

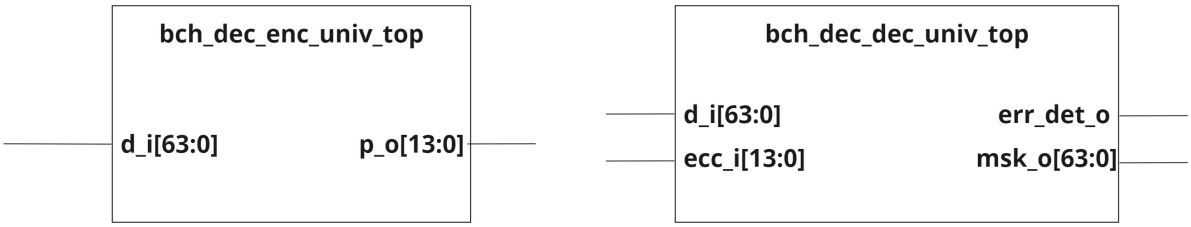


Figure 6.9: ECC encoder and decoder modules for DEC BCH codes

After including the ECC implemented version of DPRAM on all the internal memories, the results obtained from the SA fault simulation are shown in Table 6.5. As seen from the results, there are no faults which are observed at the functional outputs and not detected by the Safety Mechanism (ON faults). Therefore, the Diagnostic Coverage achieved by ECC is 100% on SA faults, as expected for a correctly implemented ECC solution capable of resolving one-bit errors. Faults categorized as Not Observed Not Diagnosed (NN) and Not Controllable (NC) result from insufficient toggling on different memory bits within the AutoSoC application. While the application can toggle a significant number of bits among the 300,000-odd potential memory locations, it does not exercise all bits, leading to the NN and NC classifications. If there were any issues with the ECC encoding and decoding functionality, faults would not have been detected at the corresponding checker outputs for the signals that were toggling. Therefore, these faults can be considered Safe. Ideally, additional tests could cover the remaining non-toggling bits. However, due to the absence of a proper randomized testbench, developing software applications specifically for toggling all memory bits and exercising fault injection is beyond the scope of this thesis.

[51] **does not consider transient fault analysis for the final evaluation of the AutoSoC.** In this study, transient faults are also injected on all memory locations, albeit on a smaller subset of the fault space and only at single timestamps. This is done because of the large number of faults in the transient fault space of memory locations. The corresponding results are shown in Table 6.6.

As seen from Table 6.6, the number of detected faults decreases in comparison to the results from

Table 6.5: Summary of SA faults on internal memories with ECC

Memory sub blocks	ND	NN/NC	Total
Fetch Instruction Cache TAG RAM	10515	10477	20992
Fetch Instruction Cache RAM	159453	102691	262144
Fetch Instructions MMU RAM	0	8192	8192
Load/Store Data Cache TAG RAM	9996	9972	19968
Load/Store Data Cache RAM	204755	57389	262144
Load/Store Store buffer	39158	12554	51712
Load/Store Data MMU RAM	0	8192	8192
Register File	5259	7029	12288
Total	429136	216496	645632
Diagnostic Coverage	**100%**		

Table 6.6: Summary of transient faults on internal memories with ECC

Memory sub blocks	ND	NN/NC	**IX**	Total
Fetch Instruction Cache TAG RAM	7954	13038	0	20992
Fetch Instruction Cache RAM	119832	142312	0	262144
Fetch Instructions MMU RAM	0	0	8192	8192
Load/Store Data Cache TAG RAM	8105	11863	0	19968
Load/Store Data Cache RAM	176504	85640	0	262144
Load/Store Store buffer	30458	21254	0	51712
Load/Store Data MMU RAM	0	0	8192	8192
Register File	3807	8481	0	12288
Total	346660	282588	16384	645632
Diagnostic Coverage	**95.48%**			

SA faults. This is attributed to transient fault injection and hold times, which might not necessarily result in fault detection or activation at all timestamps. Further, we also see that there are 16384 faults which are labeled as IX because of which **Diagnostic Coverage decreases from 100% to 95.48%**. These faults correspond to Translation Lookaside Buffer memories inside Memory Management Unit (MMU). The MMU for both instruction and data modules are not enabled and hence the signals in the particular module are held at an unknown state. However, since these memories are also instantiated as ECC-enabled instances of the same modules used in other memories, it is reasonable to expect ECC detection for this memory as well. Consequently, we can utilize the fault classification script to designate these signals as Safe (or detected), effectively achieving a Diagnostic Coverage of 100% once more.

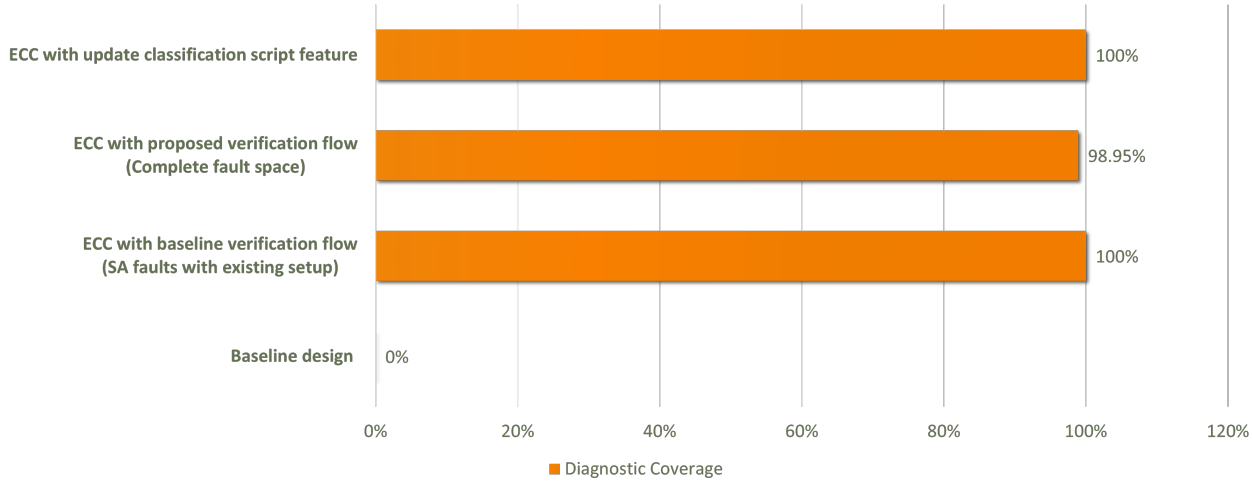


Figure 6.10: Summary of Diagnostic Coverage with ECC

A manual examination of transient faults further demonstrates that ECC errors, if they occur, will be detected regardless of the fault injection time and hold time. This is attributed to the fully combinational nature of the ECC encoder and decoder modules. Therefore, during memory read operations, any bit flips that occur at any moment can be captured by the decoder module. The final results of the ECC implementation, and its comparisons to the baseline (verification setup in the benchmark suite using XFS) and proposed flows with the update classification script are illustrated in Figure 6.10. In the next section, we delve into the logic faults within the design and explore potential enhancements to address them.

6.3.2. Results of Pipeline stage duplication with temporal redundancy

The next step is to identify areas of improvement within the logical part of the CPU. Figure 6.11 and Table 6.7 illustrate the distribution of faults on the CPU excluding the memory faults. It is evident from the figure that the Control, LSU, and Fetch modules encompass the highest percentage of faults, approximately 70%. Thus, we look at covering this fault space with the help of an additional Safety Mechanism. We apply the concept of redundancy on these modules in a way similar to DCLS to be able to achieve high diagnostic coverage with minimal effort.

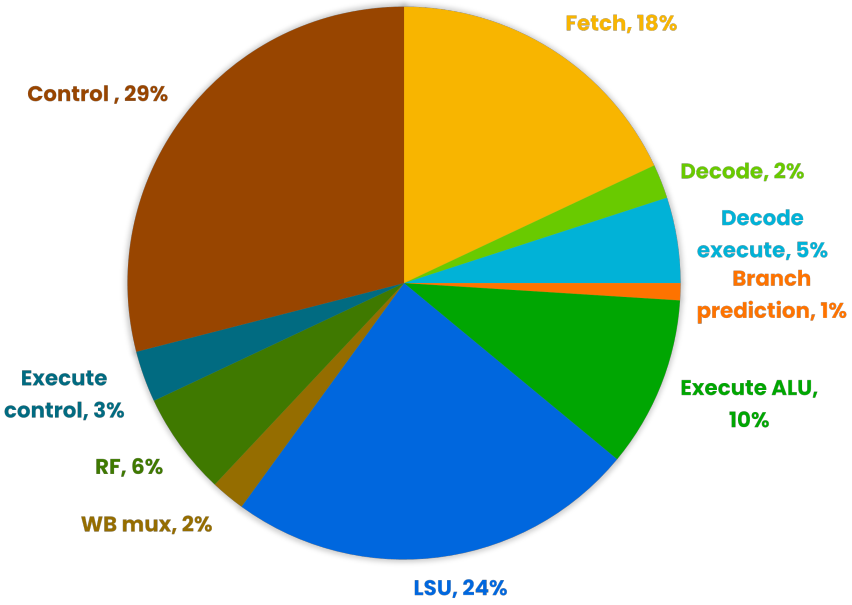


Figure 6.11: Division of logic faults

Table 6.7: Distribution of logic faults in different modules

Module	No. of faults	% of faults in CPU
Fetch	**4746**	**18%**
Decode	548	2%
Decode execute	1202	5%
Branch prediction	134	1%
Execute ALU	2716	10%
LSU	**6330**	**24%**
Wishbone (WB) mux	396	2%
Register File (RF)	1628	6%
Execute control	810	3%
Control	**7726**	**29%**
Total	26236	100%

As shown in Figure 6.12, the fetch module is duplicated with two instances, where output signals from the primary fetch are provided to the SoC (without any delay). Inputs to the shadow fetch undergo a two clock cycle delay. Similarly, outputs from the main fetch are delayed by two clock cycles for comparison with the redundant module. Any differences in outputs trigger the fetch lockstep error signal, which also functions as the checker strobe for this configuration.

With respect to the defines, another configuration, namely OPTION_FETCH_DUP, is introduced. When enabled, both instances of the fetch module are instantiated; otherwise, only a single instance is created. To assess the initial diagnostic efficiency of this Safety Mechanism, faults are injected into the reduced fault space using XFS, and the outcomes are summarized in Table 6.8.

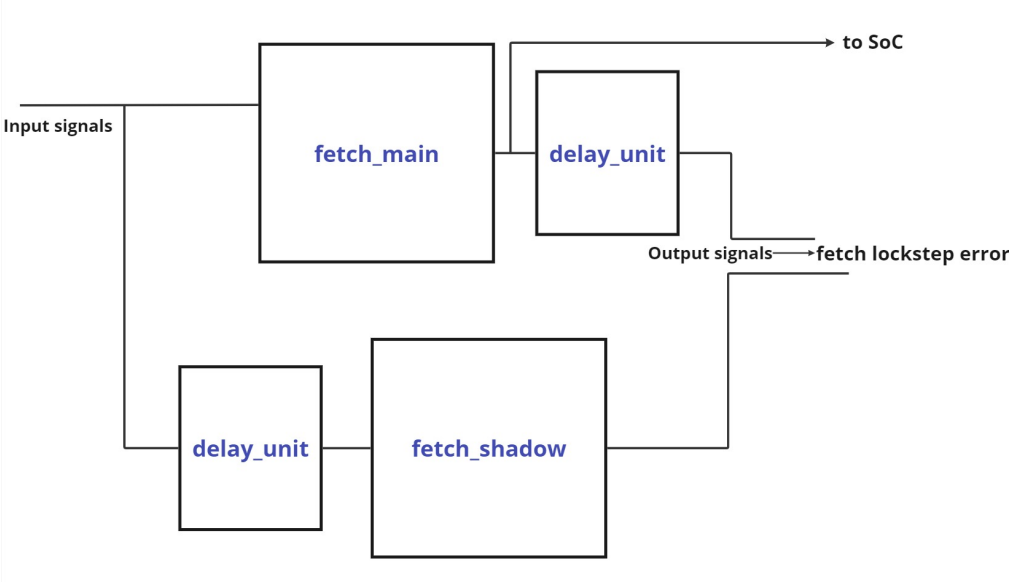


Figure 6.12: Duplication of fetch module with temporal redundancy

Table 6.8: SA Classification results of fetch duplication with time redundancy

Classification	Number of faults
Not Observed Diagnosed (ND)	579
Not Observed Not Diagnosed (NN)	3151
Observed Diagnosed (OD)	1078
Observed Not Diagnosed (ON)	4
Total	4746
Diagnostic Coverage	**99.75%**

As seen from Table 6.8, **majority of the faults inside the fetch module are detected, leading to a Diagnostic Coverage of 99.75%.** However, we see that 4 faults are observed at the functional output, but not detected by the lockstep mechanism. These four faults are residual faults observed on signals that cause a jump to an invalid PC address. For example, injecting a SA fault on `decode_branch_i` (one of the four ON signals) triggers a jump to a PC address because of the injected fault, causing an invalid PC address. Consequently, one or more of the output signals from the main CPU are held at a high impedance or unknown state. As shown in Figure 6.13, the `ibus_addr_o` signal is held at an unknown state (and subsequently `ibus_addr_o_delay`), and even though it differs from the `ibus_adr_o_shadow` signal, it still does not trigger the lockstep error, since comparisons are not valid on 'x' signals. The four faults categorized as ON are all representative of this type of fault and hence are not detected by the Safety Mechanism.

It is important to note here that the summary of faults presented in Table 6.8 cover the faults injected only on the main module. It does not cover faults injected on the redundant logic and the comparator. Faults injected solely on the safety mechanism logic can be considered Safe, since they will not affect the functional outputs of the fetch module, and will be detected by the comparator

Figure 6.13: Analysis of ON fault for fetch duplication

logic. Additionally, a manual analysis of faults on the comparator is further presented to understand the effects of comparator faults on a dual-logic setup. For instance, let us take the example of a fault injected into one of the locations inside the fetch module:

`autosoc_tb.dut.mor1kx0.mor1kx_cpu.cappuccino.mor1kx_cpu.mor1kx_fetch_cappuccino.ic_enable`

This fault triggers the fetch lockstep error signal, which occurs due to the discrepancy of the delay and shadow version of the two signals - `ibus_addr_delay_[2]` and `ibus_req_o_delay`, as shown in Figure 6.14. Given that we are only dealing with the inputs and outputs of the comparator, our focus lies with potential faults in these areas. If a SA fault is present on any one of the comparator inputs (delay or shadow signal), the error will still be detected because mismatches will be found for other timestamps. However, if both the delay and shadow signals are stuck at the same value, then the error will not be detected.

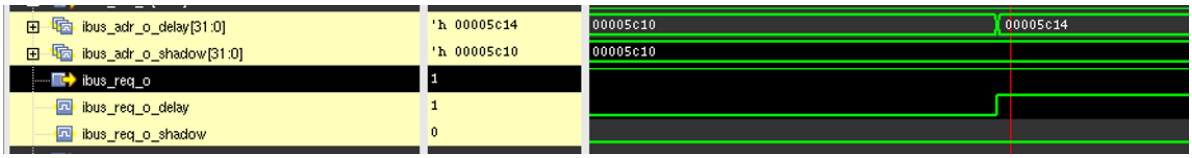


Figure 6.14: Signal mismatches from the comparator

In the event of a transient fault causing the comparator logic to fail, one such possibility is that both the inputs `ibus_adr_o_delay[2]` and `ibus_req_o_delay` have a transient fault at the exact same time and for a fixed duration. However, the probability of such a scenario might be very low as well. In most cases, the fetch lockstep error will be detected if transient faults are injected at other timestamps, or not on specific bits of the signals.

Next, we extend the dual logic design to both the Control and the LSU modules, the results of which are tabulated in Table 6.9. As shown, **LSU module duplication achieves a DC of 98.42%**, with 40 faults categorized as ON. Again, this is because of output signals being held at an unknown value and failing in comparison with the corresponding shadow values. On the other hand, **Control module duplication is able to achieve a DC of 100%**, with no residual faults.

Table 6.9: SA Classification results for duplication of Control and LSU modules

Classification	LSU	Control
Not Observed Diagnosed	1019	3635
Not Observed Not Diagnosed	3310	3212
Observed Diagnosed	1914	235
Observed Not Diagnosed	47	0
Untestable/Safe faults	40	0
Total	6330	7726
Diagnostic Coverage	**98.42%**	**100%**

We combine the implementation of ECC along with the duplication of Control, Fetch and LSU modules to obtain results arising from the overall enhancements. As shown in Table 6.10, the results include SA faults injected at all logic modules in the CPU along with internal memories. Faults are however not injected on duplicated modules and comparator logic. The numbers presented do not take into account the update classification feature and provide the baseline results on the AutoSoC design. **The final Diagnostic Coverage obtained is 97.66%, which corresponds to an ASIL C level component.** Further, it is also seen that the combination of LSU and Control module duplication also contributes to a Diagnostic Coverage of 97.33%, indicating its suitability for an ASIL C component.

Table 6.10: Overall AutoSoC classification results for SA faults (baseline flow)

Classification	Numbers
Not Observed Diagnosed	434361
Not Observed Not Diagnosed	228707
Observed Diagnosed	3227
Observed Not Diagnosed	7301
Untestable/Safe faults	684
Total	674280
Diagnostic Coverage	**98.36%**

It is important to note that these results do not account for the effect of transient faults or the influence of port and wire faults on duplicated modules, which could potentially result in different classifications. Thus, Table 6.10 presents the results of the proposed verification flow considering transient faults as well as the extended fault space. This does not include the usage of the update classification script to change fault classes of signals. **As seen, the Diagnostic Coverage decreases to 95.88%, which makes the design an ASIL B component.**

The reduction in DC can be attributed to the inclusion of transient faults along with faults injected at various location types. Common input signals shared among redundant modules will classify all faults as ON due to common-mode errors, leading to a reduced DC. Moreover, there is an increase in the number of IX faults resulting from faults injected randomly during simulation. In this group, there are 16,384 faults that can be updated to Safe/Detected status, as depicted in Figure 6.10. Using the update classification script, we modify the status of these signals, **resulting in a final Diagnostic Coverage of 97.79%, thereby reinstating it as an ASIL C component.** The final results are illustrated in Figure 6.15.

Table 6.11: Overall AutoSoC classification results for entire fault space (proposed flow)

Classification	Numbers
Not Observed Diagnosed	793387
Not Observed Not Diagnosed	43329
Observed Diagnosed	11816
Observed Not Diagnosed	13537
Untestable/Safe faults	2736
Impossible X-State	21034
Total	885839
Diagnostic Coverage	**95.88%**

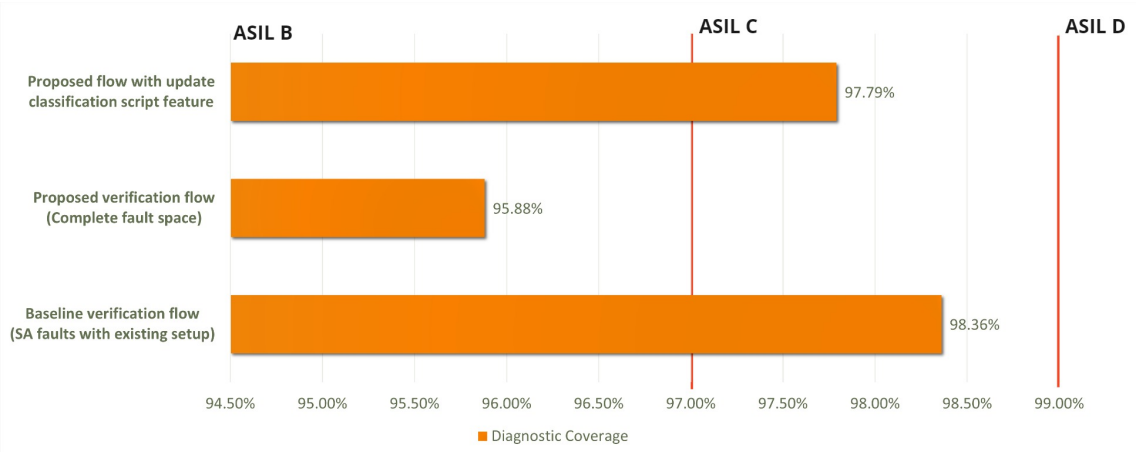


Figure 6.15: Comparison of results on AutoSoC - baseline v/s proposed flow

While the final Diagnostic Coverage remains lower compared to the baseline flow, the ASIL level of the component remained unchanged after using the proposed flow. **However, in a scenario where the Diagnostic Coverage approaches the lower limit of the ASIL metrics, the proposed flow could play a critical role in determining the ASIL of the desired component. Neglecting to consider all faults may result in the eventual metric incorrectly assigning a higher ASIL level to a component.** This could have dire implications on the final quality of the product.

6.4. Discussion

The results of the proposed verification flow on the FIFO and AutoSoC designs indicate that the Diagnostic Coverage varies depending on the faults considered for simulation. By incorporating a combination of two tools, the proposed methodology covers the maximum possible fault space, thereby providing more accurate coverage metrics. Initial analysis of the design identified areas for improvement within the memories and logic parts of the CPU, highlighting the distribution of faults in each category. Subsequently, the implementation of ECC on internal memories and duplication of three modules was performed. Following the verification flow, the AutoSoC design achieved an ASIL C certification. Now, the question arises: How do these modifications impact the area utilization of the overall design compared to the baseline design?

To estimate the additional area required for the enhancements made to the design, we consider the total number of faults. There are a total of 322,816 locations for internal memories and 13,118 logic locations. Considering that every 32 bits of data required 12 bits of ECC logic, we would need an additional 121,056 bits for ECC. Additionally, we would require extra logic for error mask, enable, encoder, and decoder logic. This is estimated by instrumenting a fault list on the RAM instance, which yields an additional 3512 locations for the ECC logic on all RAM instances. For the duplication of modules, we consider the number of fault locations duplicated in the logic, along with the comparator faults. This adds an additional 10,924 fault locations to the picture. Therefore, the estimate of the total additional fault locations in the enhanced design is 135,492, **indicating an increase of 1.4x compared to the original design area**. In comparison to DCLS, which would have a 2x increase in area while providing an ASIL D design with 99% coverage, this design does not incur as much area overhead while still achieving a coverage close to ASIL D rating.

The Diagnostic Coverage of the design could potentially improve further depending on the manual analysis of the IX faults occurring in the logic of the CPU. The presence of many signals held at an unknown state during simulation also highlights the importance of ensuring proper signal assignment for different scenarios within the design. Moreover, this situation highlights a critical issue related to transient fault injection. The potential fault space for transient faults is extensive, implying that faults could be injected at any time during simulation and for any duration. Although it was feasible to inject multiple transient faults in the FIFO design, achieving the same in a much larger and more complex design such as AutoSoC is nearly impossible. As a result, we could only inject one transient fault per location at a random timestamp. However, does the transient fault injection guarantee that a fault will be propagated at that particular timestamp, or does it have to be injected at a fixed time in order to see the effect of the fault on the strobes? The complexity of the transient fault space thus raises serious questions for future methodologies to be developed for improving estimation of transient fault space coverage.

7

Conclusion

7.1. Summary

The primary aim of the thesis was to evaluate EDA tools used for Functional Safety Verification with RTL designs, aiming to identify possible discrepancies in their outcomes and feature sets. Inconsistencies in results could potentially lead to varying ASILs being assigned to different components. Therefore, a thorough analysis of FuSa EDA tools was required to understand any disparities in fault classification results and to identify potential advantages of one tool over another, if there were any. Therefore, the primary research question was centered around comparing the results of FuSa EDA tools on reference RTL designs. Further, how could these differences be resolved to develop a better verification methodology and also provide accurate coverage metrics? The thesis aimed at addressing these concerns by comparing two FuSa EDA tools and subsequently devising a verification framework informed by the obtained results.

Chapters 1 and 2 provided initial insights by exploring the fundamentals of Functional Safety and delving into the ISO 26262 lifecycle. The problem statement and essential background information were introduced, setting the foundation for the remainder of the thesis. In Chapter 3, existing methodologies and techniques for Functional Safety Verification were analysed, alongside an examination of typical Safety Mechanisms employed in functional safe designs. This analysis highlighted the need for a comprehensive evaluation of FuSa EDA tools due to observed discrepancies in verification results across different methodologies. Chapter 4 presented a qualitative and quantitative comparison of two EDA tools, with a specific focus on their respective strengths and weaknesses. Building upon this comparative analysis, Chapter 5 proposed a novel verification methodology that combined the capabilities of both tools to develop a robust and reliable verification solution. The results of this approach were presented in Chapter 6, which scrutinized its efficacy on two distinct designs: FIFO with ECC and AutoSoC, an automotive-grade SoC tailored for functional safety applications. Additionally, Safety Mechanisms were proposed and implemented on the AutoSoC to improve its ASIL level, aligning with industry standards and requirements.

The comparison of the two EDA tools - XFS and VC Z01X provided several noteworthy insights. First, XFS had an issue of back propagation while injecting faults on input and output ports. While this serves as a valid fault scenario, it was not covering faults injected solely on the port and not back propagating. This resulted in missing a particular section of faults in XFS. VC Z01X, on the other hand, injected faults on various location types, which allowed it to exercise different types of faults. This difference had an effect on the fault classifications, thereby leading to divergent Diagnostic Coverage numbers. Second, VC Z01X boasted a notably faster fault simulation capability, attributed to its concurrent engine, in contrast to XFS. Additionally, features such as fault sampling contributed to expediting fault campaigns within VC Z01X. XFS, on the other hand, had limitations and issues with the concurrent engine, whilst not supporting SET faults and various HDL constructs. This forced usage of the serial engine, and even though runs could be split to execute multiple fault injection commands, it required usage of multiple licenses for the same. Lastly, VC Z01X lacked the capability to inject transient faults on inputs and intermediate nets, thus leaving a portion of the fault space uncovered. Despite outperforming XFS in several aspects, this limitation in VC Z01X highlighted the necessity for

comprehensive fault coverage across all fault types.

Building upon the findings of the comparison, a new verification methodology is proposed combining the two tools. First, all supported SA and transient faults are injected utilizing VC Z01X. Next, post-processing scripts are employed to extract reports identifying faults that were not injected with VC Z01X. For these remaining signals, fault injection commands are automatically generated using scripts for execution with XFS. This combined approach ensures comprehensive coverage of the maximum possible fault space, minimizing the risk of overlooking any faults and thereby enhancing the accuracy of Diagnostic Coverage metrics. Moreover, the methodology capitalizes on the faster fault simulation flow of VC Z01X while strategically leveraging XFS to cover the unexplored fault space, thus optimizing efficiency and efficacy in fault simulation.

The proposed verification flow was tested on two designs - one being a reference example of FIFO enabled with ECC, which also served as the basis of comparison between the two tools. The second design was AutoSoC, an open-source benchmark suite featuring configurable Safety Mechanisms. The results revealed that the final Diagnostic Coverage differed from the individual results obtained from each tool. However, the metric derived from the combined flow offered a more accurate depiction of the fault space, aiming to minimize oversights in injected faults. Upon initial evaluation of AutoSoC utilizing the verification flow, potential areas for enhancement were identified. Subsequently, additional Safety Mechanisms were integrated, including ECC implementation on internal memories and pipeline stage duplication with temporal redundancy. After utilising the update classification feature of the proposed verification flow combined with these enhancements, a final Diagnostic Coverage of 97.79% was achieved. This resulted in an ASIL C rating for the design, while incurring an estimated area increase of 1.4x. In comparison to DCLS, which boasts a Diagnostic Coverage of 99% and a 2x increase in area, this design exhibits favorable performance in terms of area utilization while achieving coverage close to an ASIL D rating.

7.2. Limitations and Future Work

While the proposed flow carefully aims to consider the maximum possible fault space attainable by the two tools, the results also highlight a critical issue in transient fault campaigns. The possibility of injecting transient faults at different timestamps with varying hold times presents quite an extensive transient fault space. However, for a large and complex design, it might not be possible to inject all such faults. Even if we consider a portion of these faults, is it possible to make sure that the injected faults will result in propagation to the concerned outputs? If not, how do we ensure that the injection and hold times are selected in a manner that guarantees the faults will indeed impact the strobes? While these questions could not be answered in the current scope of this thesis, they pave the way for future methodologies to consider these aspects of the transient fault space for better coverage estimation. Further, extensive guidelines regarding the injection of transient faults and the associated diagnostic coverage should be incorporated into the ISO standard.

An enhancement to the proposed verification methodology could involve the automated generation of tests for faults undetected by existing testbenches and scenarios. Drawing inspiration from ATPG tools, which generate inputs to propagate faults to desired outputs, automated tests could be developed. However, devising tests at a higher abstraction level such as RTL poses several challenges compared to GLN, where fault propagation paths are more evident. Nevertheless, there remains potential to develop tests for faults categorized as “Not Controllable (NC)”, for instance, by toggling the relevant signals in additional tests. These automated tests could reduce the manual workload on verification engineers, speeding up fault simulation campaigns and ultimately resulting in accurate coverage results.

Bibliography

[1] M. Placek, *Automotive electronics cost as a percentage of total car cost worldwide from 1970 to 2030*. [Online]. Available: https://www.statista.com/statistics/277931/automotive-electronics-cost-as-a-share-of-total-car-cost-worldwide/, (accessed: Jan 30, 2023).

[2] N. Kaitwade, *Automotive Electronics Market Trend and Forecast by 2033 | FMI*. [Online]. Available: https://www.futuremarketinsights.com/reports/automotive-electronics-market, (accessed: Feb 16, 2023).

[3] *Automotive electronics market - global industry analysis, size, share, growth, trends analysis, regional outlook and forecasts, 2023 - 2032*. [Online]. Available: https://www.precedenceresearch.com/automotive-electronics-market, (accessed: Jan 30, 2023).

[4] *Automotive electronics market size, share and trends analysis report by component (electronic control unit, sensors, current carrying devices), by application, by sales channel, by region, and segment forecasts, 2023 - 2030*. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/automotive-electronics-industry#, (accessed: Nov 29, 2023).

[5] M. Placek, *Share of vehicles recalled due to electronic components defect worldwide between 2011 and 2019*. [Online]. Available: https://www.statista.com/statistics/1276588/share-of-vehicles-recalled-due-to-electronic-components-defect-worldwide/, (accessed: Jan 6 2023).

[6] "ISO 26262 Road Vehicles - Functional Safety," International Organization for Standardization, Dec. 2018.

[7] *IEC 61508 Functional Safety Standard*. [Online]. Available: https://www.tuvsud.com/en-us/services/functional-safety/iec-61508, (accessed: Nov 29, 2023).

[8] K.-L. Lu, Y.-Y. Chen, and L.-R. Huang, "FMEDA-based fault injection and data analysis in compliance with ISO-26262," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2018, pp. 275–278. DOI: 10.1109/DSN-W.2018.00075.

[9] F. A. d. Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Efficient methodology for ISO26262 functional safety verification," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 255–256. DOI: 10.1109/IOLTS.2019.8854449.

[10] A. C. Bagbaba, F. A. d. Silva, S. Hamdioui, and C. Sauer, "Combining fault analysis technologies for ISO26262 functional safety verification," in *2019 IEEE 28th Asian Test Symposium (ATS)*, 2019, pp. 129–1295. DOI: 10.1109/ATS47505.2019.00024.

[11] A. Sherer, J. Rose, and R. Oddone, "Ensuring functional safety compliance for ISO 26262," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–3. DOI: 10.1145/2744769.2747924.

[12] R. Debouk, *Overview of the second edition of ISO 26262: Functional safety— road vehicles*. [Online]. Available: https://jsystemsafety.com/index.php/jss/article/view/55/52, (accessed: Jan 15, 2023).

[13] *"Management of Functional Safety: Part 2 of ISO 26262"*. [Online]. Available: https://www.kuglermaag.com/functional-safety/management-of-functional-safety-iso-26262/, (accessed: Jan 18, 2023).

[14] D. Howard, "Functional safety and engineering design automation," in *2019 Pan Pacific Microelectronics Symposium (Pan Pacific)*, 2019, pp. 1–9. DOI: 10.23919/PanPacific.2019.8696578.

[15] A. Sarkar, *"ASIL (automotive safety integrity levels) ratings simplified"*. [Online]. Available: https://www.linkedin.com/pulse/asil-automotive-safety-integrity-levels-ratings-animesh-sarkar?_l=en_US, (accessed: Jan 2023).

[16] S. Zhou and S. Zhang, "Study on stability control during split-mu ABS braking," in *2011 Chinese Control and Decision Conference (CCDC)*, 2011, pp. 1235–1239. DOI: 10.1109/CCDC.2011.5968377.

[17] *Introduction – What is Functional Safety? – ABLIC Inc.* [Online]. Available: https://www.ablic.com/en/semicon/products/automotive/asil/, (accessed: Jan 24, 2023).

[18] "ISO 26262 Road Vehicles - Functional Safety, Part 5: Product development at the hardware level," International Organization for Standardization, Oct. 2018.

[19] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 970–975. DOI: 10.1109/ICCAD.2017.8203886.

[20] D. Alexandrescu, A. Evans, M. Glorieux, and I. Nofal, "EDA support for functional safety — how static and dynamic failure analysis can improve productivity in the assessment of functional safety," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017, pp. 145–150. DOI: 10.1109/IOLTS.2017.8046210.

[21] O. Ballan, P. Maillard, J. Arver, *et al.*, "Evaluation of ISO 26262 and IEC 61508 metrics for transient faults of a multi-processor system-on-chip through radiation testing," *Microelectronics Reliability*, vol. 107, Apr. 2020. DOI: 10.1016/j.microrel.2020.113601.

[22] J. Stoppe, R. Wille, and R. Drechsler, "Cone of influence analysis at the electronic system level using machine learning," in *2013 Euromicro Conference on Digital System Design*, 2013, pp. 582–587. DOI: 10.1109/DSD.2013.69.

[23] *VC Formal User Guide*, version U-2023.03-SP2, Synopsys.

[24] *JasperGold Functional Safety Verification App User Guide*, version 2021.12, Cadence.

[25] M. Gros, *When and how to qualify tools according to ISO 26262*. [Online]. Available: https://www.btc-embedded.com/when-and-how-to-qualify-tools-according-to-iso-26262/, (accessed: Jan 26, 2023).

[26] V. Prasanth, D. Foley, and S. Ravi, "Demystifying automotive safety and security for semiconductor developer," in *2017 IEEE International Test Conference (ITC)*, 2017, pp. 1–10. DOI: 10.1109/TEST.2017.8242074.

[27] S.-H. Jeon, J.-H. Cho, Y. Jung, S. Park, and T.-M. Han, "Automotive hardware development according to ISO 26262," in *13th International Conference on Advanced Communication Technology (ICACT2011)*, 2011, pp. 588–592.

[28] Y.-C. Chang, L.-R. Huang, H.-C. Liu, C.-J. Yang, and C.-T. Chiu, "Assessing automotive functional safety microprocessor with iso 26262 hardware requirements," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, 2014, pp. 1–4. DOI: 10.1109/VLSI-DAT.2014.6834876.

[29] O. Ballan, U. Rossi, A. Wantens, J.-M. Daveau, S. Nappi, and P. Roche, "Verification of soft error detection mechanism through fault injection on hardware emulation platform," in *2010 International Conference on Dependable System and Networks Workshops (DSN-W), Jun*, 2010.

[30] M. Kooli and G. D. Natale, "A survey on simulation-based fault injection tools for complex systems," May 2014, pp. 1–6, ISBN: 978-1-4799-4972-4. DOI: 10.1109/DTIS.2014.6850649.

[31] F. Ferlini, L. O. Seman, and E. A. Bezerra, "Enabling ISO 26262 compliance with accelerated diagnostic coverage assessment," *Electronics*, vol. 9, no. 5, 2020, ISSN: 2079-9292. [Online]. Available: https://www.mdpi.com/2079-9292/9/5/732.

[32] C. Lopez-Ongil, M. Garcia-Valderas, M. Portela-Garcia, and L. Entrena, "Autonomous fault emulation: A new FPGA-Based acceleration system for hardness evaluation," *IEEE Transactions on Nuclear Science*, vol. 54, no. 1, pp. 252–261, 2007. DOI: 10.1109/TNS.2006.889115.

[33] A. Cagri Bagbaba, F. Augusto Da Silva, and C. Sauer, "Improving the Confidence Level in Functional Safety Simulation Tools for ISO 26262," in *2018 Design and Verification Conference Europe*, Zenodo, 2018. DOI: 10.5281/zenodo.3361607.

[34] F. Augusto Da Silva, A. Cagri Bagbaba, S. Hamdioui, and C. Sauer, "Use of Formal Methods for verification and optimization of Fault Lists in the scope of ISO26262," in *2018 Design and Verification Conference Europe*, Zenodo, 2018. DOI: 10.5281/zenodo.3361533.

[35] A. Cagri Bagbaba, F. A. da Silva, S. Hamdioui, and C. Sauer, "An automated formal-based approach for reducing undetected faults in ISO 26262 hardware compliant designs," in *2021 IEEE International Test Conference (ITC)*, 2021, pp. 329–333. DOI: 10.1109/ITC50571.2021.00047.

[36] F. A. da Silva, A. C. Bagbaba, S. Sartoni, *et al.*, "Determined-safe faults identification: A step towards ISO26262 hardware compliant designs," in *2020 IEEE European Test Symposium (ETS)*, 2020, pp. 1–6. DOI: 10.1109/ETS48528.2020.9131568.

[37] J. E. R. Condia, F. A. Da Silva, S. Hamdioui, C. Sauer, and M. S. Reorda, "Untestable faults identification in GPGPUs for safety-critical applications," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2019, pp. 570–573. DOI: 10.1109/ICECS46596.2019.8964677.

[38] *Cortex-M33 Dual Core Lockstep, version 1.0*, 2017.

[39] P. Li, W. Xi, X. Jiang, and Q. Chen, "Fault-tolerant design of power edge computing processor based on full-hardware dual-core lockstep," in *2021 IEEE 5th Conference on Energy Internet and Energy System Integration (EI2)*, 2021, pp. 2398–2403. DOI: 10.1109/EI252483.2021.9712894.

[40] F. Abate, L. Sterpone, C. A. Lisboa, L. Carro, and M. Violante, "New techniques for improving the performance of the lockstep architecture for SEEs mitigation in FPGA embedded processors," *IEEE Transactions on Nuclear Science*, vol. 56, no. 4, pp. 1992–2000, 2009. DOI: 10.1109/TNS.2009.2013237.

[41] M. T. Sim and Y. Zhuang, "A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications," in *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*, 2020, pp. 2231–2238. DOI: 10.1109/IECON43393.2020.9255188.

[42] M. Santhiya, S. Saranya, S. Vijayachitra, C. B. Lavanya, and M. Rajarajeswari, "Application of voter insertion algorithm for fault management using triple modular redundancy (TMR) technique," in *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, 2021, pp. 578–583. DOI: 10.1109/ICICV50876.2021.9388450.

[43] K. A. Al-Kofahi, *Reliability analysis of triple modular redundancy system with spare*, Thesis, Available at https://repository.rit.edu/theses, 1993.

[44] S.-W. Kwak and K.-H. You, "Reliability analysis and fault tolerance strategy of TMR real-time control systems," *Journal of Institute of Control, Robotics and Systems*, vol. 10, Jan. 2004. DOI: 10.5302/J.ICROS.2004.10.8.748.

[45] N. Li, E. Dubrova, and G. Carlsson, "Evaluation of alternative LBIST flows: A case study," in *2014 NORCHIP*, 2014, pp. 1–5. DOI: 10.1109/NORCHIP.2014.7004708.

[46] F. Brglez, G. Gloster, and G. Kedem, "Built-in self-test with weighted random pattern hardware," in *Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1990, pp. 161–166. DOI: 10.1109/ICCD.1990.130190.

[47] H. J. Wunderlich and G. Kiefer, "Bit-flipping BIST," IEEE, 1996, pp. 337–343. DOI: 10.1109/iccad.1996.569803.

[48] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An efficient Bist scheme based on reseeding of multiple polynomial linear feedback shift registers," in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, pp. 572–577. DOI: 10.1109/ICCAD.1993.580117.

[49] A.-W. Hakmi, S. Holst, H.-J. Wunderlich, J. Schlöffel, F. Hapke, and A. Glowatz, "Restrict encoding for mixed-mode BIST," in *2009 27th IEEE VLSI Test Symposium*, 2009, pp. 179–184. DOI: 10.1109/VTS.2009.43.

[50] P. Bernardi, R. Cantoro, A. Floridia, *et al.*, "Non-intrusive self-test library for automotive critical applications: Constraints and solutions," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 920–923. DOI: 10.23919/DATE.2019.8714780.

[51] F. A. da Silva, A. Cagri Bagbaba, A. Ruospo, *et al.*, "Special session: AutoSoC - a suite of open-source automotive SoC benchmarks," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, 2020, pp. 1–9. DOI: 10.1109/VTS48691.2020.9107599.

[52] *AutoSoC benchmark suite*. [Online]. Available: https://www.autosoc.org/home, (accessed: Jan 13, 2023).

[53] *OpenRISC*. [Online]. Available: https://github.com/openrisc, (accessed: Feb 8, 2023).

[54] *RTEMS Real Time Operating System (RTOS)*. [Online]. Available: https://www.rtems.org/, (accessed: Jan 2023).

[55] R. Tervo, *Introduction to BCH codes*. [Online]. Available: https://www.ece.unb.ca/tervo/ece4253/bch.shtml, (accessed: Feb 29, 2023).