Eda tools and methodologies for reliable nanoelectronic systems

Augusto da Silva, F.

DOI
10.4233/uuid:92bc5717-c27d-4cd2-b2b2-560c6551e437

Publication date
2022

Document Version
Final published version

Citation (APA)
Augusto da Silva, F. (2022). *Eda tools and methodologies for reliable nanoelectronic systems*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:92bc5717-c27d-4cd2-b2b2-560c6551e437

EDA TOOLS AND METHODOLOGIES FOR RELIABLE NANOELECTRONIC SYSTEMS

EDA TOOLS AND METHODOLOGIES FOR RELIABLE NANOELECTRONIC SYSTEMS

Dissertation

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Wednesday 21 September 2022 at 17:30 o'clock

by

Felipe AUGUSTO DA SILVA

Master of Science in Electrical Engineering,
Federal University of Santa Catarina, Brazil
born in São Paulo, Brazil.

This dissertation has been approved by the promotors.

promotor:	Prof. dr. ir. S. Hamdioui
copromotor:	Dr. ir. J.S.S.M. Wang

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. ir. S. Hamdioui	Delft University of Technology, promotor
Dr. ir. J.S.S.M. Wang	Delft University of Technology, copromotor
Independent members:	
Dr. M. Jenihhin	Tallinn University of Technology, Estonia
Prof. dr. ir. G. Gaydadjiev	University of Groningen, the Netherlands
Prof.dr. ir. M. Sonza Reorda	Politecnico di Torino, Italy
Prof. dr. ir. A.J. van der Veen	Delft University of Technology
Dr. C. Sauer	Cadence Design Systems, Germany
Prof. dr. ir. R.E. Kooij	Delft University of Technology, reserve member

Keywords: Functional Safety, Verification, ISO 26262, Fault Space Analysis, Tool Qualification, Fault Injection Simulation, Formal Methods, Automotive benchmark, Safe Faults, Software Test Library, Safety Metrics, SPFM, ASIL

Printed by: Ipskamp Printing, the Netherlands

Front & Back: design by Felipe Augusto da Silva

ISBN 978-94-6366-596-4

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

Education is the most powerful
weapon you can use to change the world.

Nelson Mandela

CONTENTS

SUMMARY

In recent years, advances in technology have enabled the employment of automated systems to control driving tasks. The idea of electronic devices having complete control over a vehicle promises to change the concept of mobility soon. However, allowing computers to control all the tasks in a vehicle demands sophisticated systems and significant safety concerns. Furthermore, the increasing complexity in such applications is causing a shift in the traditional design flow. For example, the development of semiconductors implementing safety-critical functionalities must incorporate mechanisms to reduce the risk of failures avoiding life-threatening situations. This dissertation addresses the role of the EDA industry in supporting the safety aspects of automotive electronic systems. We propose methodologies to deploy the traditional EDA technologies into functional safety verification, improving compliance to Automotive Safety Standards, like ISO 26262, and ensuring automotive devices' safety integrity levels. For such, we must comprehend how the guidelines of ISO 26262 establish a comprehensive safety lifecycle that supports the analysis of *Systematic Failures* and *Random Hardware Failures*. Afterward, we investigate the many possibilities to advance the state-of-the-art by deploying EDA technologies in compliance with safety requirements. As a result, we identify research possibilities at different safety lifecycle stages. Furthermore, we propose methodologies to support such development phases, enabling compliance with ISO 26262.

Initially, we need to ensure that the software tools deployed in the development of safety-critical systems have the necessary levels of confidence; in other words, a malfunction in such software tools cannot mask or fail to identify failures in the design. For such, we propose a methodology that deploys multiple technologies capable of classifying the behavior of faults; if one of the tools has a malfunction, the classifications will differ, revealing a possibility of safety violation. Furthermore, the execution of the methodology is controlled by an application that automates the analysis and generates a report highlighting any discrepancies.

Another crucial aspect for validating any methodology targeting automotive safety is the presence of representative test cases. Nonetheless, the limited access to representative designs and industrial methodologies poses a challenge to the research community. Therefore, we present the AutoSoC Benchmark Suite. The AutoSoC intends to provide researchers with an automotive SoC based on commercial solutions, including all essential components, highly customizable and allowing comparability between distinct methodologies and results. Furthermore, this dissertation describes the conception, features, and configurations targeting compliance with different levels of automotive safety.

Even though safety requirements are established earlier in automotive systems' development, the safety metrics verification is only possible at later stages of the lifecycle; failing to achieve the required figures demands additional iterations causing a high impact on costs and development time. For that reason, this work proposes a systematic approach for the early estimation of safety metrics of Automotive designs. The method-

ology is based on the characterization of the design description (RTL and gate-level) and the workload impact concerning fault propagation. Ultimately, the gathered information allows an early assessment of the Safety Mechanisms' performance at various development phases; and, therefore, it will enable the early estimation of safety metrics.

Later, during the development of safety features, engineers encounter a typical trade-off between cost and safety. The conventional safety schemes, such as Dual-Core Lock-Step (DCLS), require full redundancy of the hardware area, increasing costs. We propose a methodology that automatically generates test patterns for CPUs, avoiding hardware overhead. First, the process deploys formal verification to analyze the propagation of faults in a CPU; the analysis results in a sequence of test stimuli required to propagate the fault. Then, by integrating such sequences as a Software Test Library (STL), we can replicate this behavior for detecting faults during FI Simulation. Such an approach results in a standalone safety mechanism enabling detection of faults and increasing the safety integrity of the CPU without additional hardware.

Furthermore, we propose a methodology to support the final clauses for hardware safety verification, enabling the validation of the contributions to previous development phases and assessing the safety integrity level of the target SoC. For such, we introduce an automated approach for identifying the nature of faults not concerning safety-critical functionalities. The severe demands for tolerance to random faults demand a comprehensive fault space analysis. As part of this process, fault classification methods are still driven by experts, requiring manual analysis that is very expensive, time-consuming, and prone to errors. The proposed methodology begins with code coverage analysis for identifying design elements where a fault cannot disturb safety-critical functionalities. Next, those elements are automatically translated into formal rules and configured in a formal analysis environment, enabling the identification of additional Safe faults.

Finally, we confirm this dissertation's contributions to the safety lifecycle by completing ISO 26262 hardware verification clauses assuming the proposed methodologies. As a result, the final figures show supplementary coverage in the implemented Safety Mechanisms, improving the safety metrics and enabling compliance with ASIL C requirements. Also, the results satisfy the validation of our test case, enabling an accurate safety evaluation permitting compliance to ISO 26262 without hardware redundancy.

SAMENVATTING

Dankzij de technologische ontwikkelingen van de laatste jaren is het mogelijk geautomatiseerde systemen te gebruiken om het rijden van voertuigen te besturen. Het idee dat elektronische apparaten de volledige controle over een voertuig hebben, belooft het concept van mobiliteit binnenkort te veranderen. Computers alle taken in een voertuig laten uitvoeren vereist echter gesofisticeerde systemen en aanzienlijke veiligheidsrisico's. Bovendien leidt de toenemende complexiteit van dergelijke toepassingen tot een verschuiving in de traditionele ontwerpprocessen. Zo moeten bij de ontwikkeling van halfgeleiders die veiligheidskritische functies uitvoeren, mechanismen worden ingebouwd om het risico van falen te verminderen en zo levensbedreigende situaties te vermijden. In dit proefschrift wordt ingegaan op de rol van de EDA-industrie bij de ondersteuning van de veiligheidsaspecten van elektronische systemen voor de automobielindustrie. Wij stellen methodologieën voor om traditionele EDA-technologieën te gebruiken bij de verificatie van functionele veiligheid en zo de naleving van veiligheidsnormen voor auto's, zoals ISO 26262, te verbeteren en de veiligheidsintegriteit van auto-elektronica te verzekeren. Daartoe moeten wij begrijpen hoe de richtlijnen van ISO 26262 een uitgebreide veiligheidslevenscyclus vaststellen die de analyse van systematische storingen en willekeurige hardwarefouten ondersteunt. Vervolgens onderzoeken wij de vele mogelijkheden om de stand van de techniek te verbeteren door EDA-technologieën te gebruiken die aan de veiligheidseisen voldoen. Als gevolg daarvan identificeren wij onderzoeksmogelijkheden in verschillende stadia van de veiligheidslevenscyclus. Verder stellen wij methodologieën voor om die ontwikkelingsstadia te ondersteunen, teneinde te voldoen aan ISO 26262.

In eerste instantie moeten wij ervoor zorgen dat de software-instrumenten die bij de ontwikkeling van veiligheidskritische systemen worden gebruikt, de nodige betrouwbaarheidsniveaus hebben; met andere woorden, een storing in die software-instrumenten kan geen fouten in het ontwerp maskeren of niet aan het licht brengen. Daarom stellen wij een methode voor waarbij meerdere technologieën worden gebruikt die het gedrag van fouten kunnen classificeren; als een van de hulpmiddelen een storing heeft, zullen de classificaties verschillen, wat een mogelijke veiligheidsovertreding aan het licht brengt. Bovendien wordt de uitvoering van de methodologie gecontroleerd door een toepassing die de analyse automatiseert en een rapport genereert waarin eventuele afwijkingen worden aangegeven.

Een ander essentieel aspect voor de validering van een methodologie die gericht is op de veiligheid van auto's is de aanwezigheid van representatieve testgevallen. De beperkte toegang tot representatieve ontwerpen en industriële methodologieën vormt echter een uitdaging voor de onderzoeksgemeenschap. Daarom stellen wij de AutoSoC Benchmark Suite voor. De AutoSoC is bedoeld om onderzoekers een SoC voor auto's te bieden die gebaseerd is op commerciële oplossingen, met inbegrip van alle essentiële componenten, zeer aanpasbaar is en vergelijkbaarheid tussen verschillende methodologieën en

resultaten mogelijk maakt. Verder beschrijft dit proefschrift het ontwerp, de kenmerken, en de configuraties die gericht zijn op de naleving van verschillende veiligheidsniveaus voor auto's.

Hoewel de veiligheidsvoorschriften reeds in een vroeg stadium bij de ontwikkeling van automobielsystemen worden vastgesteld, is de verificatie van de veiligheidscijfers pas in latere stadia van de levenscyclus mogelijk; als men er niet in slaagt de vereiste cijfers te halen, zijn extra iteraties nodig, wat een grote invloed heeft op de kosten en de ontwikkelingstijd. Daarom wordt in dit werk een systematische aanpak voorgesteld voor de vroege schatting van veiligheidscijfers van automobielontwerpen. De methodologie is gebaseerd op de karakterisering van de ontwerpbeschrijving (RTL en gate-niveau) en de werklast die de voortplanting van fouten met zich meebrengt. Uiteindelijk maakt de verzamelde informatie een vroege beoordeling mogelijk van de prestaties van de veiligheidsmechanismen in de verschillende ontwikkelingsfasen; en daardoor zal het mogelijk zijn de veiligheidscijfers vroegtijdig te schatten.

In een later stadium, bij de ontwikkeling van veiligheidsfuncties, stuiten de ingenieurs op een typische afweging tussen kosten en veiligheid. De conventionele veiligheidssystemen, zoals Dual-Core LockStep (DCLS), vereisen volledige redundantie van de hardware, waardoor de kosten stijgen. Wij stellen een methode voor die automatisch testpatronen voor CPU's genereert, waarbij hardware-overhead vermeden wordt. Eerst gebruikt het proces formele verificatie om de voortplanting van fouten in een CPU te analyseren; de analyse resulteert in een opeenvolging van teststimuli die nodig zijn om de fout voort te planten. Door deze opeenvolgingen vervolgens in een Software Test Library (STL) te integreren, kunnen wij dit gedrag repliceren voor het opsporen van fouten tijdens FI Simulatie. Een dergelijke aanpak resulteert in een stand-alone veiligheidsmechanisme waarmee fouten kunnen worden opgespoord en de veiligheidsintegriteit van de CPU kan worden verhoogd zonder extra hardware.

Ook stellen wij een methode voor om de definitieve bepalingen voor de verificatie van de hardwareveiligheid te ondersteunen, zodat de bijdragen aan de vorige ontwikkelingsfasen kunnen worden gevalideerd en het veiligheidsintegriteitsniveau van de beoogde SoC kan worden beoordeeld. Daartoe introduceren wij een geautomatiseerde aanpak voor het identificeren van de aard van fouten die geen betrekking hebben op veiligheidskritieke functies. De strenge eisen die gesteld worden aan de tolerantie ten opzichte van toevallige fouten vereisen een uitgebreide foutenruimte-analyse. Als onderdeel van dit proces worden foutclassificatiemethoden nog steeds door deskundigen gestuurd, zodat handmatige analyse nodig is, die zeer duur, tijdrovend en foutgevoelig is. De voorgestelde methode begint met een analyse van de codedekking, om te bepalen in welke ontwerpelementen een fout de veiligheidskritische functies niet kan verstoren. Vervolgens worden die elementen automatisch vertaald in formele regels en geconfigureerd in een formele analyse-omgeving, waardoor bijkomende veilige fouten kunnen worden geïdentificeerd.

Tenslotte bevestigen wij de bijdragen van dit proefschrift aan de veiligheidslevenscyclus door de ISO 26262 hardwareverificatieclausules te voltooien, uitgaande van de voorgestelde methodologieën. Het resultaat is dat de eindcijfers aanvullende dekking laten zien in de geïmplementeerde veiligheidsmechanismen, waardoor de veiligheidsmetrieken verbeteren en aan de ASIL C-eisen kan worden voldaan. Ook voldoen de resultaten

aan de validatie van onze testcase, waardoor een nauwkeurige veiligheidsevaluatie mogelijk wordt, die naleving van ISO 26262 zonder hardware-redundantie mogelijk maakt.

1

INTRODUCTION

The prospect of fully autonomous vehicles promises to revolutionize mobility concepts in the coming years. As a result, the semiconductors industry invests heavily to support the technologies embedded in a vehicle. Nonetheless, the concept of computers having complete control of a car implies new challenges, as a life-threatening situation caused by a malfunction in electronic systems is unacceptable. Therefore, this dissertation addresses the role of the EDA industry in supporting the safety aspects of automotive electronic systems. We start with a comprehensive investigation of the challenges imposed by the automotive safety standards, e.g., ISO 26262; we describe the requirements for the concept, production, verification, and supporting process for the development of automotive semiconductors. Next, we discuss the challenges for compliance with safety requirements and their relation to the EDA technologies. For each listed challenge, we investigate the state-of-the-art, considering the methods applied by the automotive industry and the contributions from academia. After, we describe the research topics explored during this Ph.D. project. Then, we present this dissertation's main contribution; we propose methodologies that deploy the traditional EDA technologies into innovative solutions to ensure the safety integrity levels of automotive devices. Finally, we detail the thesis organization.

1

1.1. MOTIVATION

In recent years, the development of electronic systems has become an essential asset in the revenue of the automotive sector. Established applications, as Advanced Driver Assistance Systems (ADAS) and recent trends, as Hybrid/Electric Vehicles (HEV/EV), have a relevant influence on the sector's success. Figure 1.1 illustrates a study from the IHS Markit denoting the increase of the average semiconductor value per car and its impact on the overall revenue for the sector. The study outlines a Compound Annual Growth Rate (CAGR) of 7,1% with a relevant increase in some application domains. The ADAS, for example, has the potential to grow around 19% per year in the following years. This application already represents a share of six billion dollars, potentially reaching eight billion dollars shortly.

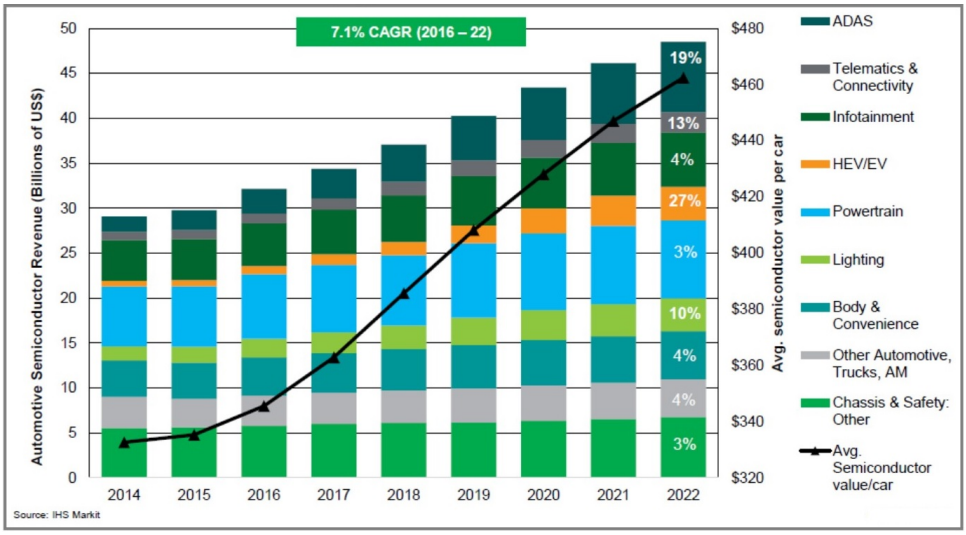

Figure 1.1: Automotive semiconductor revenue by application

This favorable scenario generates momentum for investments aiming to advance the technologies embedded in a vehicle. All the supply chain associated with the development of automotive solutions has their share of contribution. When considering semiconductors, one of the crucial shareholders is the Electronic Design Automation (EDA) industry. EDA tools are essential for developing semiconductors and have supported the advances in technology for several years. Nowadays, EDA companies provide several products to assist the traditional semiconductors development flow and tailored solutions to address the needs of specific industries, such as Aerospace, Defense, Hyperscale Computing, 5G Systems, and Automotive. Nonetheless, the advances in automotive technologies are increasing the burden of electronic systems to new levels, and the traditional solutions do not suffice to attend to the sector's requirements. Furthermore, the concept of autonomous vehicle applications implies new challenges, as a life-threatening situation caused by a malfunction in electronic systems is unacceptable.

The idea of electronic devices having complete control over a vehicle promises to

change the concept of mobility. However, allowing computers to control all the tasks in an automobile requires high complexity systems and significant safety concerns. Also, the development of autonomous vehicles applications, where a system failure could cause life-threatening situations, entails state-of-the-art challenges for reducing the chances of such shortcomings. Therefore, compliance with Functional Safety Standards is of high priority. The primary standard for the automotive sector is the "ISO 26262 Road vehicles – Functional safety" [1]. The standard defines requirements to assure that functional safety is a fundamental part of developing automotive electronic systems; and describes measures to confirm that the probability of failures is reduced to acceptable levels. Compliance with ISO 26262 is an arduous task and should be addressed by all supply-chain involved in developing automotive solutions, including the EDA industry.

This research addresses the role of the EDA industry in supporting the safety of automotive electronic systems. We propose tools and methodologies to deploy the traditional EDA technologies into functional safety verification, improve compliance to ISO 26262, and ensure the safety integrity levels of automotive devices.

1.2. FUNCTIONAL SAFETY VERIFICATION BY ISO 26262

Functional safety aims to reduce the risk of physical injury and damage to health caused by failures in safety-critical systems; it requires implementing protection (safety) measures to reduce the risk of such events. In other words, Functional safety aims to assure that a system will perform its intended function correctly. In case of a failure, the system will fail in a predictable (safe) manner. There are specific functional safety standards for several industries, such as Defense, Aviation, Aerospace, and Medical. As cited before, the primary standard for the automotive sector is the ISO 26262.

The first edition of ISO 26262, targeting passenger cars, was published in 2011. A second edition, published in 2018, extended the scope to all road vehicles. The standard is considered a best practice framework for achieving automotive functional safety. The requirements from ISO 26262 cover the entire product lifecycle, ranging from specification, design, implementation, integration, verification, validation, and production release. Similar to others, ISO 26262 is a risk-based safety standard; it defines quantitative and qualitative measures to assess the risk of hazardous situations. Then, safety measures should be implemented to control or mitigate the effects of failures that could cause such hazards. The implementation of safety measures must consider the class of the failure they aim to mitigate. ISO 26262 defines two categories of failures: *Systematic Failures* and *Random Hardware Failures*.

- *Systematic Failures* are failures generally caused by human errors in defining, designing, configuring, installing, calibrating, and maintaining a system; good practices and quality control should recognize such failures. ISO 26262 defines this class of failures as "failure related in a deterministic way to a specific cause, that can only be eliminated by a change of the design or manufacturing process, operational procedures, documentation or other relevant factors." If, for example, the implementation of a function is incorrect due to a misinterpretation of a requirement, such failure should be identified by a development flow that contemplates requirement-based testing and review.

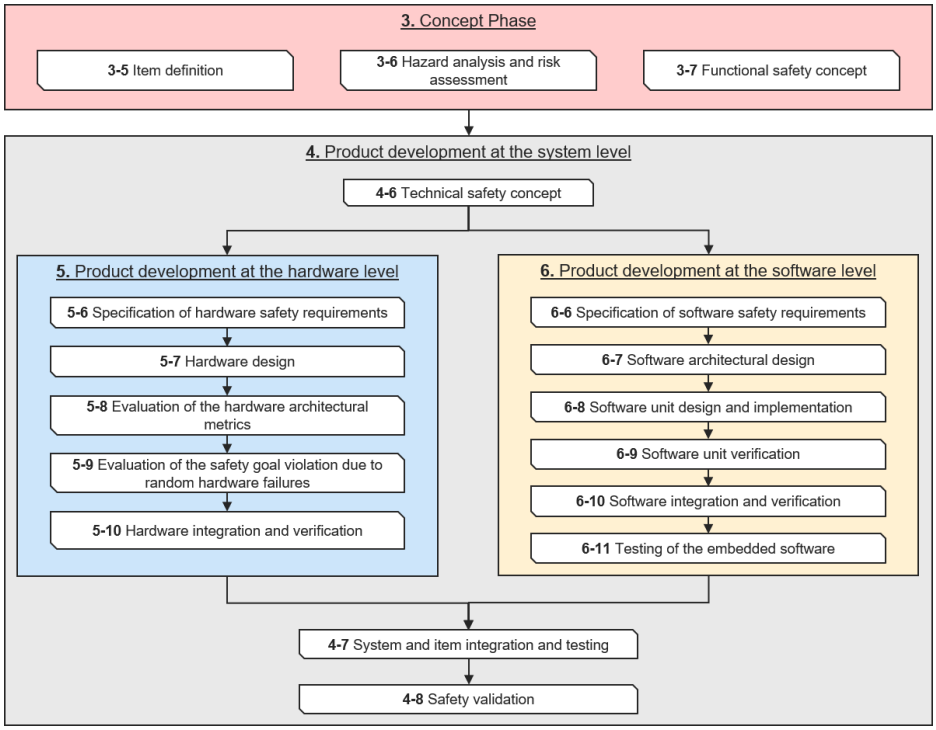

Figure 1.2: ISO 26262 Safety Lifecycle

- *Random Hardware Failures*: are failures in which the cause is not always observable or controllable; such failures have a probability of happening in designs without apparent errors, and their root cause is usually not noticeable. According to ISO 26262 definition, these are "failures that can occur unpredictably during the lifetime of a hardware element, and that follows a probability distribution." Common causes of *Random Hardware Failures* are defects inherent in the technology process, usage condition, aging, interference, radiation, among others. As root causes are not predictable, the failure rate of random faults cannot generally be reduced; thus, the focus is on handling these faults employing detection and correction mechanisms.

ISO 26262 defines guidelines and means to access the quality of the implemented measures for both failure categories. For *Random Hardware Failures*, the standard defines safety metrics to assess if the probabilistic risk is sufficiently low. Meanwhile, the control of *Systematic Failures* is achieved by the safety lifecycle defined by the standard. The safety lifecycle devises a product life in three phases: Concept, Product Development, and After the release for production. Each phase has determined clauses, objectives, and work-products, including well-defined pass criteria between the clauses. Each clause also defines verification requirements for the work-products, assuring the required quality levels are achieved. Figure 1.2 illustrates the ISO 26262 Safety Lifecycle.

In the following sections, we describe the main elements of the Concept and Product Development phases of the safety lifecycle. Since this thesis is primarily concerned with the EDA role in safety hardware development, we will focus on the requirements related to such parts.

1.2.1. CONCEPT PHASE

The *Concept Phase* elaborates initial impressions and definitions as the functional concept for the product development. It also establishes the safety plan, with the activities to ensure safety during the development process and tailoring its actions accordingly. Consequently, the safety plan determines the overall sequence of steps to assure a safety development process and compliance to ISO 26262. Additionally, the *Concept Phase* includes clauses to identify and categorize hazard situations resulting from malfunctions and determine the required safety integrity level during the development. This phase defines the clauses: *Item Definition, Hazard Analysis and Risk Assessment* and *Functional Safety Concept.* Figure 1.3 illustrates the clauses of the *Concept Phase.*

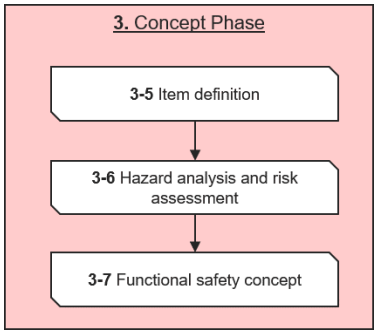

Figure 1.3: Concept Phase

ITEM DEFINITION

As defined by ISO 26262, an item is a system or array of systems to implement a function at the vehicle level. In other words, an item relates to the specific functionality under development. The *Item Definition* clause defines and describes the item functionalities and dependencies; also, it determines the item interactions with the driver, the environment, and other items at the vehicle level. Another objective is to support an adequate understanding of the item with a high-level description of its limitations and intended functionalities; such a clause enables a better comprehension for subsequent phases.

HAZARD ANALYSIS AND RISK ASSESSMENT

This clause aims to identify and categorize possible hazards triggered by failures in the item defined in the previous clauses; for each hazard, we need to formulate safety goals for prevention and mitigation, avoiding unreasonable risks to human life. The *Hazard Analysis and Risk Assessment (HARA)* is a structured method for understanding what might go wrong with a system and the resulting hazards.

1

The ISO 26262 defines a hazard as *"a potential source of harm caused by malfunctioning behavior of the item."* The potential harm of an item malfunction depends on circumstances where the malfunction occurs. Hence, the identification of possible hazards is based on the analysis of operational situations. For instance, the analysis of a malfunction in an Anti-lock Braking System (ABS) must consider operational situations of the vehicle; the hazard resulting from such a malfunction is different in a parking vehicle or a vehicle at high speed on a highway. Therefore, the combination of a hazard and an operational situation defines a hazardous event. As an example, possible hazardous events resulting from a malfunction in an ABS could be:

- Loss of stability when the breaks are activated during a sudden stop on a highway.
- Loss of stability when the breaks are activated during a sudden stop in an unpaved road.
- Unintended activation of the breaks in the city center.
- Unintended activation of the breaks on a highway.

A comprehensive list of hazardous events must be prepared based on the vehicle class, where the system will be employed, and the outputs from the *Item Definition* clause. Next, all hazardous events need to be classified by three metrics Exposure, Severity, and Controllability.

Exposure defines how likely is the hazard to happen; it must consider factual information regarding similar vehicles and systems, traffic accidents rate, and the employed technologies. The Severity represents the hazardous event's harmful; it examines the possible level of impairment to human lives. Finally, the Controllability assesses the possibility of the driver to control the situation and avoid the hazardous event. Figure 1.4 illustrates the possible classes of Exposure, Severity, and Controllability.

Exposure		Severity		Controllability	
E0	Incredible	S0	No injuries	C0	Controllable in general
E1	Very low probability	S1	Light and moderate injuries	C1	Simply controllable
E2	Low probability	S2	Severe and life-threatening injuries (survival probable)	C2	Normally controllable
E3	Medium probability	S3	Life-threatening injuries (survival uncertain), fatal injuries	C3	Difficult to control or uncontrollable
E4	High probability				

Figure 1.4: Exposure, Severity, and Controllability classes for Risk Analysis

The combined classification of these metrics defines the Automotive Safety Integrity Level (ASIL) allocation. The ASIL establishes the strictness of the safety requirements to be respected during the development of automotive components. According to ISO 26262, there are four ASILs classes, ranging from A to D; ASIL A represents the lower grade in the safety spectrum, while ASIL D symbolizes systems with the highest degree of hazard risk. As the ASIL allocation is based on the probability and acceptability of harm, the rigor applied to safety assurance depends on the system's functionality. For example,

systems like airbags and ABS require an ASIL D because the risks associated with their failure are the highest. On the other end, components like infotainment may only require ASIL A. Other examples are headlights and brake lights that generally would be ASIL B; or cruise control systems that would frequently be ASIL C. Figure 1.5 demonstrates the ASIL allocation based on the Exposure, Severity, and Controllability classification. When the risk of harm is sufficiently low, the hazardous event is classified as Quality Management level (QM); such classification represents not dangerous components that do not dictate any safety requirements to be managed under the ISO 26262.

		Exposure	Controllability		
			C1	C2	C3
Severity	S1	E1	QM	QM	QM
		E2	QM	QM	QM
		E3	QM	QM	ASIL A
		E4	QM	ASIL A	ASIL B
	S2	E1	QM	QM	QM
		E2	QM	QM	ASIL A
		E3	QM	ASIL A	ASIL B
		E4	ASIL A	ASIL B	ASIL C
	S3	E1	QM	QM	ASIL A
		E2	QM	ASIL A	ASIL B
		E3	ASIL A	ASIL B	ASIL C
		E4	ASIL B	ASIL C	ASIL D

Figure 1.5: Automotive Safety Integrity Level (ASIL) allocation

The ASIL allocation is determined by how much threat the malfunctioning of a particular component can cause under various situations. Consequently, it also defines the demands for safety processes and the level of risk reduction needed to achieve a tolerable risk. Therefore, throughout the ISO 26262 safety lifecycle, the ASIL will determine the requirements of each clause. Higher ASILs demand additional verification steps improving the product quality and reducing the risk of *Systematic Failures*. The ASIL allocation also defines the required measures for controlling *Random Hardware Failures*. The standard defines metrics to evaluate the effectiveness of implemented measures to cope with such failures. The requirements for *Random Hardware Failures* are defined in the *Product Development at Hardware level* section.

At this stage, risk reduction is exemplified as a Safety Goal. According to ISO 26262, safety goals are the first level of safety requirements. They are high-level descriptions that will be detailed into procedures to bring the vehicle to a safe state in case of malfunctions. Therefore, during the HARA, each analyzed hazardous event will have an allocated ASIL and derived safety goals. If, for example, we consider the situation below resulting from a malfunction in an ABS component:

- Unintended activation of the breaks on a highway.

Such a hazardous event can have a high Exposure (E4), considering the usage of vehicles on highways; moreover, Controllability may be difficult (C3) if the car is at high speeds;

finally, Severity is very high (S3) in this situation. This example would result in an ASIL D allocation. For such, the safety goals listed below could be derived:

- The ABS must not limit the functionality of the breaking system.
- The ABS must include mechanisms to avoid unintended activation of the breaks.

Under the context of functional safety, the safety goals are more critical than the functionalities of the automotive systems; one of the main objectives of the safety lifecycle is to avoid violation of these goals. In the following development phases, the safety goals will be derived into a *Functional Safety Concept* and later into requirements for the Software and Hardware components that will fulfill the system.

FUNCTIONAL SAFETY CONCEPT

The objective of this clause is to derive the functional safety requirements from the *Item Definition* and the safety goals; then, these requirements must be allocated to the primary architectural elements, defining provisions for software and hardware development. The *Functional Safety Concept* also contemplates the high-level definition of safety mechanisms. Therefore, the functional safety requirements must envision mechanisms to avoid safety goals violations. Conventional descriptions employed at this stage of development are:

- Fault detection,
- Mitigation,
- Safe states,
- Warnings to the driver,
- Redundancies.

The work-products of the *Functional Safety Concept* clause will enable the initiation of product development at the system, software, and hardware levels.

1.2.2. PRODUCT DEVELOPMENT AT SYSTEM LEVEL

The objective of the Product Development phases is to elaborate the work-products from the Concept Phase into a concrete implementation, including confirmation of the safety concept. This phase starts with the System development and continues with Hardware and Software development; for each, the standard defines a v-model approach, where work-products are verified to assure compliance to safety requirements. In addition, each clause includes pre-requisites ensuring that it only starts when the previous clauses are concluded and verified. Figure 1.6 illustrates the *Product Development at system level* phase; it also highlights the clauses responsible for verification of past clauses work-products.

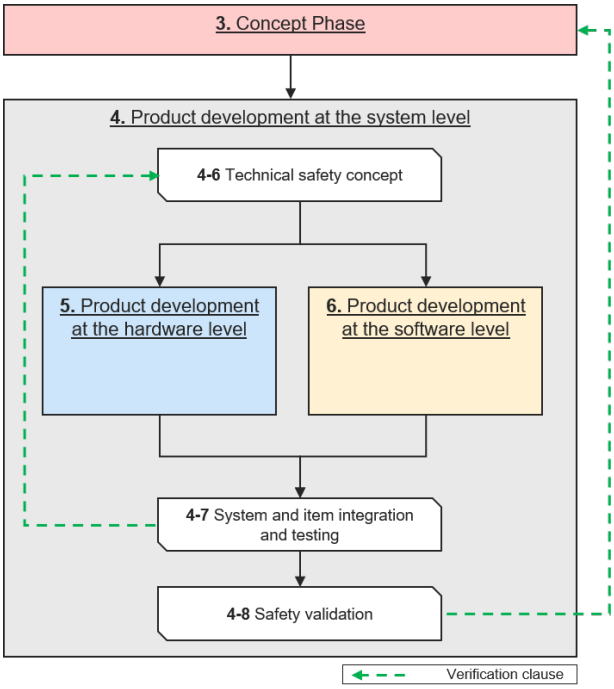

Figure 1.6: Product Development at the system level

TECHNICAL SAFETY CONCEPT

The *Product Development at system level* starts with the *Technical Safety Concept*. This clause aims to refine the functional safety concept into technical requirements. Consequently, the system's functionalities must be elaborated, including the definition of the system architecture, dependencies and properties of the system elements, interfaces between internal and external components, and allocation of requirements to hardware and software. In addition, it contemplates the specification of safety mechanisms, fault detection intervals, and the transition to the Safe States.

At this stage, after the elaboration of the hardware architecture, it is possible to establish the measures to avoid *Random Hardware Failures*; for that, safety engineers must estimate, for each hardware component, the probability of failures (Failure Rate). Next, they must define the requirements of Diagnostic Coverage to reduce the Failure Rate to acceptable levels. The Diagnostic Coverage represents the percentage of faults detected by Safety Mechanisms. Traditional methods to improve the accuracy of this analysis are Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA). The confirmation of the measures to avoid *Random Hardware Failures* made at this stage will only happen at the end of the *Product Development at hardware level*. For that reason, the accuracy of this analysis is of high importance; failing to achieve the estimated metrics will entail modifications at later development stages.

1

SYSTEM AND ITEM INTEGRATION AND TESTING

The work-products of the *Technical Safety Concept* allow the start of the product development at software and hardware levels. Still, verifying the product at the system level is only possible after software and hardware are available. Therefore, the *System and item integration and testing* clause can only start after sufficient completion of hardware and software development. This clause comprises integration tests of the hardware and software elements, subsystems, and other in-vehicle systems. The objectives of the *System and item integration and testing* are to demonstrate that the defined safety measures, resulting from system safety analyses, are correctly implemented; and to provide evidence that the integrated system elements fulfill their safety requirements.

SAFETY VALIDATION

Finally, the final clause of the *Product Development at system level* aims to verify that the system implementation fulfills the definitions from the *Concept Phase*. The *Safety validation* clauses provide evidence that the safety goals are achieved when integrated into the vehicle(s); and that the functional and technical safety concepts are appropriate for achieving functional safety.

1.2.3. PRODUCT DEVELOPMENT AT HARDWARE LEVEL

The hardware development must fulfill the definitions from the *Technical Safety Concept*; the work-products of the before-mentioned clause determine the expected hardware behavior. Examples of such are the Technical safety concept, System architectural design specification, and Hardware-software interface specification. Figure 1.7 illustrates the clauses for the development of hardware.

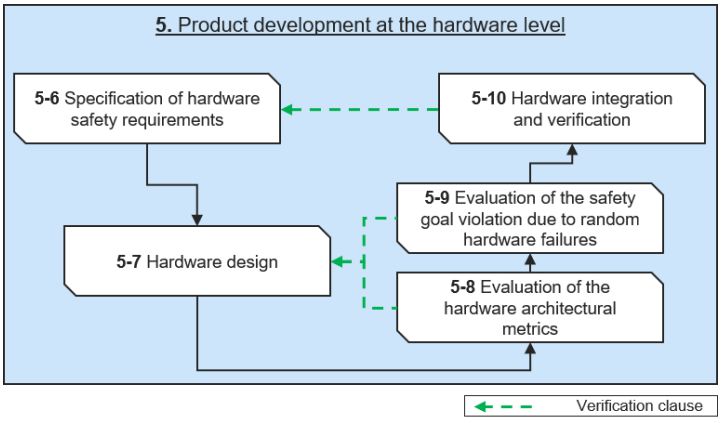

Figure 1.7: Product Development at the hardware level

SPECIFICATION OF HARDWARE SAFETY REQUIREMENTS

The *Product development at hardware level* starts with further refinement of the system definitions. The *Specification of hardware safety requirements* has as objectives the development of hardware safety requirements, derived from the technical safety concept

and the system architectural design; and also, the refinement of the hardware-software interface.

The definitions at the hardware level should include attributes to ensure the effectiveness of safety mechanisms and considerations over safety metrics; for such, engineers must determine fault tolerance, fault detection intervals, and the target diagnostic coverage for each safety mechanism. Also, at this stage, the hardware-software interface must include a description of safety-related dependency between hardware and software.

HARDWARE DESIGN

This clause ensures that the hardware design development complies with the system design specification and the hardware safety requirements. The *Hardware design* clauses do not determine the design cycle steps, allowing the use of standard design cycles; for example, the development lifecycle for an Integrated Circuit could be: Architectural Design, Functional and Logic Design, Circuit Design, Physical Design, Verification and Signoff, Layout Post Processing, and Fabrication. Nevertheless, the clause includes objectives aiming to guarantee that the overall safety lifecycle is respected and fulfills the safety requirements, avoiding *Systematic Failures*.

For ensuring compliance with the safety requirements, *Hardware design* clauses also determine the extension of the safety analysis started at the *Technical Safety Concept*. The elaboration of the deductive (i.e., FTA) and inductive (i.e., FMEA) analysis, at this stage, can include factual information about the implementation of the hardware. The example below represents one possible Failure Mode (FM) in the register bank of a given CPU; such FM results from elaborating a high-level failure mode that describes a Deadlock in the CPU. The FM definition starts with the FMEA during the *Technical Safety Concept* and is refined with the hardware implementation information.

- FM-1: The value in the program counter register becomes corrupted causing a deadlock.

After implementing the hardware responsible for the program counter register, we can map the FM-1 to the flops and gates performing the functionality; such mapping enables an assessment of the probability of an occurrence of FM-1. For that, we must determine the Failure Rate for each hardware element (flop, gate, and memory cell). The Failure Rate is measured in Failure In Time (FIT) units, where FIT represents the number of failures per billion hours of operation of the component. The FIT of the hardware elements is generally extracted from the history of use of a given tape-out technology; alternatively, it can be defined based on standards as the IEC 62380 Electronic Reliability Prediction Standard. Finally, by considering the FIT rate of each hardware element mapped to the FM, we can determine the Raw FIT rate of each FM. Eventually, Failure Modes should cover every hardware component implementing safety-related functionalities; the sum of the Raw FIT rate of each FM results in the total FIT rate of the hardware.

EVALUATION OF THE HARDWARE ARCHITECTURAL METRICS

This clause aims to provide evidence of the suitability of the hardware architecture design concerning detection and control of safety-related *Random Hardware Failures*. The

evaluation is the initial verification step that allows engineers to demonstrate compliance to the *Concept Phase* assumptions for *Random Hardware Failures*.

The evidence of detection and control of *Random Hardware Failures* can be shown by the hardware architectural metrics. The metrics defined by ISO 26262 are the Single-Point Fault Metric (SPFM) and the Latent Fault Metric (LFM). The SPFM represents the threats imposed by faults that Safety Mechanisms do not protect. The LFM symbolizes the danger of faults that cannot directly violate a safety goal but could be a risk in the presence of a second fault (i.e., faults in a Safety Mechanism).

To calculate the hardware architectural metrics, we need to understand the effect of faults in the hardware functional behavior and classify them accordingly. According 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 at all inputs and outputs of the design gates; the combination of all faults comprise the Fault Space. The philosophy behind it is to examine the hardware behavior under the influence of each element of the Fault Space. Then, based on the alterations caused by the fault, we can determine the fault classification in one of the sub-classes defined below.

- Safe faults: these are faults that do not cause any disturbance of safety-critical functionalities.
- Detected faults: these are faults that can disturb the safety-critical functionalities; hence Safety Mechanisms (SMs) are deployed to correct them and ensure that they become innocent.
- Undetected: these are faults for which the effect is unknown; they can be either safe, detected or even dangerous faults without associated safety mechanisms.

The results of the Fault Space analysis must confirm the assumptions related to the technical safety concept; for example, the clause should have defined Safety Mechanisms to protect the hardware from faults that affect safety-related functionalities. Additionally, the analysis confirms the effectiveness of implemented SM by demonstrating that a fault was detected, avoiding the disruption of the hardware behavior. The understanding regarding the fault classification and safety diagnostic is crucial to achieving a comprehensive functional safety analysis.

Next, we must confirm that the probability of failures is reduced to acceptable levels. For such, we need to refine the FMEA, including the results of the Fault Space analysis; the examination of Failure Modes together with safety diagnostic information is called Failure Modes Effects and Diagnostic Analysis (FMEDA). The FMEDA integrates the FMEA information, FIT of each FM based on the correlation to semiconductor components, with the analysis of faults in such components. In the FMEDA, we can incorporate for each FM the fault classification; with such, we can demonstrate that part of the FIT is related to Safe or Detected faults, reducing the residual FIT of the FM.

The example in Table 1.1 represents one FMEDA input related to the Data RAM of a given CPU; such input represents an FM in a memory that includes an Error Correction Detection Code (ECC) mechanism. Following ISO 26262 recommendations, we can claim that the ECC can correct 99% of the single faults in the memory; therefore, as most

faults are handled, only the remaining 1% must be accounted for the residual FIT. The parameters of the FMEDA, demonstrated in Table 1.1 are:

- ID: a unique identifier for the Failure Mode;
- Failure Mode: a textual description of the cause of the failure or the possible way the system can fail;
- Area: total size, in micro square meters, of the semiconductor components mapped to the FM;
- Gates: number of gates mapped to the FM;
- Flops: number of flops mapped to the FM;
- Memory cells: number of memory cells, or memory bits, mapped to the FM;
- Raw FIT rate: total FIT rate considering the probability of failures of the semiconductor components mapped to the FM;
- Safe Faults: percentage of faults, in the components mapped to the FM, classified as Safe during the Fault Space analysis;
- Safety Mechanism: identification of SMs protecting the semiconductor components mapped to the FM;
- DC: percentage of faults that are detected/corrected by the SM;
- Residual FIT rate: the final probability of failure of the FM after excluding the contribution of Safe and Detected faults.

Table 1.1: FMEDA example

ID	Failure Mode	Area	Gates	Flops	Memory cells	Raw FIT rate	Safe Faults (%)	Safety Mechanism	DC (%)	Residual FIT rate
FM_2	Data RAM corrupted value is read by the CPU leading to a wrong result	11888,2	0	0	131072	3,715	0%	ECC	99%	0,03715

The functional safety analysis of the hardware design requires the correlation of FM to all safety-related hardware components; by doing so, we can determine the Raw FIT rate and the Residual FIT rate of all Failure Modes, and therefore of all hardware components. Thus, the sum of the FIT rates of all FMs will represent the total FIT rate of the hardware design.

Considering the results of the Faults Space analysis, we understand the effect of faults on each hardware component. Therefore, we can also represent the Raw FIT rate (λ) as the sum of the contribution of all the fault classes. ISO 26262 defines the following fault classes:

- λSPF represents Single-Point faults that are not protected by SMs.
- λR describes Residual faults that are Undetected by SMs.

- λMPF or multi-point faults represent faults that could only violate a safety goal when combined with a second fault. Such class can be divided into two subclasses:
  - Detected/Perceived Multi-Point Faults (λMPF,DP) are the Detected faults that could only pose a thread in combination with a second fault.
  - Latent Multi-Point Faults (λMPF,L) are faults that can only violate a safety goal combined with a second fault (i.e., faults in a Safety Mechanism).
- λS represents the contribution of Safe faults.

The sum of the fault classes is equal to the Raw λ, as defined in the equation 6.1.

$$\lambda = \lambda SPF + \lambda R + \lambda MPF + \lambda S \tag{1.1}$$

The definition of the fault classes is necessary to calculate the hardware architectural metrics; such operation is mandatory to demonstrate the capacity of the hardware design to cope with *Random Hardware Failures*. The minimum coverage values for the Single-Point Fault Metric (SPFM) and the Latent Fault Metric (LFM) are defined based on the target ASIL. The table 1.2 describes the target values.

The SPFM examines the fault classes with direct potential to violate safety goals in case of single-point faults. As λSPF represents faults that SM does not protect, and λR describes faults that are Undetected by SM, both classes contribute significantly to the Residual λ. The equation 6.2 illustrates the method for determining the SPFM.

$$SPFM = 1 - \frac{\sum(\lambda SPF + \lambda R)}{\sum \lambda} \tag{1.2}$$

The LFM represents the coverage of faults that cannot directly violate a safety goal but could be a risk in the presence of a second fault. For such, we must subtract the contribution of single-point faults (λSPF and λR) and include the failure risk of multi-point faults. The equation 6.3 defines the LFM calculation.

$$LFM = 1 - \frac{\sum(\lambda MPF, L)}{\sum(\lambda - \lambda SPF - \lambda R)} \tag{1.3}$$

EVALUATION OF THE SAFETY GOAL VIOLATION DUE TO RANDOM HARDWARE FAILURES

This clause aims to provide evidence that the residual risk of a safety goal violation due to *Random Hardware Failures* is sufficiently low. However, even after complying with the requirements of the *Evaluation of the hardware architectural metrics* clause, it is necessary to demonstrate that the risk posed by residual faults is acceptable. For such, ISO 26262 defines two methods for evaluating the residual risk.

The first method is a comprehensive evaluation of each possible cause of safety goal violation; for such, individual analysis of each failure violating a safety goal shall be performed at the hardware part level. This evaluation shall provide evidence the risk of failures from the different fault classes is acceptable. Hence, the analysis must consider the occurrence rate of failures caused by single-point faults, residual faults, multi-point faults, and the presence of diagnostic coverage concerning detection intervals.

The second method is the calculation of the Probabilistic Metric for Random Hardware Failures (PMHF). The PMHF value represents the average probability of failure per hour over the operational lifetime of the hardware design. In other words, it demonstrates the remaining possibility of failure after all safety measures are in place. Thus, the PMHF evaluation proves that all hardware elements' cumulative safety target violating failure rate is sufficiently low. However, unlike the other metrics, ISO 26262 does not define a method to perform the PMHF analysis. For that reason, the methodology can be derived from different safety standards; we refer to the IEC 61508 for the PMHF calculation as in 6.4.

$$PMHF = \sum \lambda SPF + \sum \lambda R + \sum \lambda MPF,L \tag{1.4}$$

Table 1.2 illustrates the Safety Metrics minimum value required for each ASIL. It is essential to note that only now, at the verification clauses of the *Product development at hardware level* can engineers demonstrate compliance to the *Concept Phase* assumptions for *Random Hardware Failures*. If the hardware architecture fails to achieve the ASIL requirements, it must be modified; requiring rework of the safety life cycle steps. The calculation of the Safety Metrics is not required for hardware designs targeting ASIL A; in such cases, as the risk of failures causing life-threatening situations is negligible, there are no requirements for *Random Hardware Failure* control. For hardware designs targeting higher ASIL, compliance to the Safety Metrics values is mandatory.

Table 1.2: Safety Metrics Requirements per ASIL

ASIL	SPFM	LFM	PMHF
A	Not Relevant	Not Relevant	Not Relevant
B	>90%	>60%	<100 FIT
C	>97%	>80%	<100 FIT
D	>99%	>90%	<10 FIT

HARDWARE INTEGRATION AND VERIFICATION

This clause aims to ensure that the developed hardware complies with the hardware safety requirements and design specifications. The verification of the hardware with requirements-based tests is a common practice to assure compliance with requirements and avoid *Systematic Failures*. For such, the hardware integration test cases shall consider the requirements and interfaces, equivalence classes, boundary values, based error guessing, functional dependencies, and others. This clause must also consider the verification of the SM; for such, it is possible to reuse the functional testing and fault injection simulation results. The *Hardware integration and verification* also requires the confirmation of the robustness of hardware against external stresses, considering the operational conditions of the system.

1.2.4. SUPPORTING PROCESSES

This chapter defines guidelines and additional considerations to support engineers in accomplishing compliance to the safety lifecycle. As in the other chapters, the *Supporting processes* includes clauses with defined objectives, prerequisites, and work-products;

however, the clauses are not sequential; instead, they describe critical processes that support the entire safety lifecycle. The list below summarizes the objective of each clause.

- Interfaces within distributed developments: defines the interactions and dependencies between customers and suppliers for development activities;
- Specification and management of safety requirements: ensure the correct specification of safety requirements and its consistent management throughout the entire safety lifecycle;
- Configuration management: ensure that lifecycle products are under proper version control, can be uniquely identified, and are reproducible;
- Change management: analyze and control changes to safety-related work-products;
- Verification: ensures that the work-products comply with their requirements;
- Documentation management: aims the development of a documentation management strategy for the entire safety lifecycle;
- Confidence in the use of software tools: provides criteria to determine the required level of confidence and to create evidence that the deployed software tools are suitable to be used to support the activities required by the ISO 26262;
- Qualification of software components: provide evidence for the suitability of software components reuse in items developed in compliance with the ISO 26262;
- Evaluation of hardware elements: ensure that the functional behavior of hardware elements is adequate to meet the allocated safety requirements and therefore the risk of a violation of a safety goal is sufficiently low;
- Proven in use argument: guides the argument for the reuse of existing elements when field data is available;
- Interfacing an application that is out of scope of ISO 26262: aims to achieve confidence that such applications are not able to violate safety goals;
- Integration of safety-related systems not developed according to ISO 26262: aims to confirm that a system designed according to another standard still satisfies the required level of functional safety.

Although several clauses of the *Supporting processes* define critical activities for compliance with ISO 26262, in the next section, we are going to elaborate the *Confidence in the use of software tools* clause.

CONFIDENCE IN THE USE OF SOFTWARE TOOLS

The conception of automotive systems demands numerous software tools to support development and verification activities; tools such as compilers, simulators, synthesis engines, among others, are mandatory for complex solutions. However, as these tools

have a critical role in modifying work-products of the safety lifecycle, they could also introduce failures that can lead to the violation of safety goals. For that reason, software tools used to support the activities required by ISO 26262 must undergo an analysis process to determine the necessary level of confidence in their effectiveness.

This clause applies to any software tool that supports development activities required by ISO26262. For each software tool, engineers have to determine if a malfunction can introduce or mask errors in safety-related items; if the answer is yes, the clause provides means for the evaluation and qualification of the software tools. Furthermore, if the software tool requires qualification, its development process must be adequate concerning compliance with the ISO 26262. In other words, the development of the software tool must respect a safety lifecycle to avoid the occurrence of *Systematic Failures*.

The evaluation of the software tool's confidence depends on the safety activity the tool supports; also, its functionalities and properties. Hence, the examination aims to determine the impact caused by a malfunction, the Tool Impact (TI), and the measures implemented to detect malfunctions in the tool, the Tool error Detection (TD). The list below describes the possible values for TI and TD:

- Tool Impact (TI)
  - TI1: all other cases;
  - TI2: malfunctions can introduce or fail to detect errors in a safety-related item.
- Tool error Detection (TD)
  - TD1: high degree of confidence that a malfunction will be prevented or detected;
  - TD2: medium degree of confidence that a malfunction will be prevented or detected;
  - TD3: all other cases.

The analysis of TD and TI is crucial to assure that the outputs of software tools are reliable. For example, we can analyze the results of a logic synthesis software tool; such a tool aims to translate the Register Transfer Level (RTL) description of a hardware element into a gate-level design implementation. If the synthesis result is wrong due to a malfunction in the software tool, the intended functionality of the hardware design may be affected, leading to safety goal violations. Consequently, such an example should be classified as TI2 due to the possibility of introducing errors to the design implementation. In the specific case of the logic synthesis software tool, it is common practice that the resulting gate-level design implementation is verified employing equivalence checking. Such a method assures that the functionality implemented by the gate-level is equivalent to the RTL description. Therefore, as the logic synthesis output is verified, we can claim a high degree of confidence that a malfunction would be detected. For that reason, the example could be classified as TD1. The combination of TI and TD values determines the Tool Confidence Level (TCL), in the example above the software tool would be classified as TCL1. Table 1.3 illustrates the possible combinations and the resulting TCL.

Table 1.3: Determination of the Tool Confidence Level (TCL)

		Tool error Detection (TD)		
		TD1	**TD2**	**TD3**
Tool Impact (TI)	**TI1**	TCL1	TCL1	TCL1
	TI2	TCL1	TCL2	TCL3

The TCL classification will determine what the requirements for using the software tool are. For example, in cases where the analysis results in TCL2 and TCL3, the Tool Qualification is required. On the other hand, a software tool classified at TCL1 needs no additional qualification methods. Figure 1.8 illustrates the tool evaluation flow.

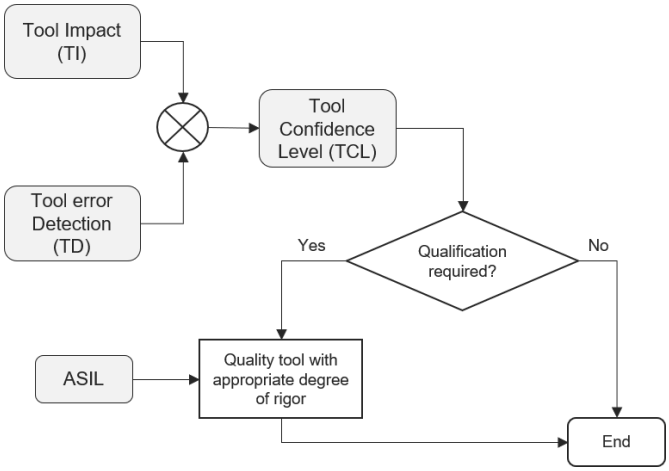

Figure 1.8: Tool Evaluation Flow

When the tool qualification is required, ISO 26262 defines methods to ensure that the risk of malfunctions is sufficiently low. The requirements are dependent on the software tool TCL and the ASIL assigned to the tool's work product. The goal of the qualification is to guarantee that a tool malfunction is not going to influence the safety functionalities; for such, engineers can deploy one of the following methods:

- Increase the confidence in the outputs of the tool employing additional verification.
- Demonstration of the tool development process following safety standards or another development process with an acceptable level of quality.
- Validation of the software tool through test and verification.

1.2.5. DISCUSSION

As described above, there are several challenges to achieving compliance with ISO 26262. The guidelines of the standard establish a comprehensive safety lifecycle that confirms

that the automotive design is free of *Systematic Failures*. Additionally, it stipulates functional safety verification requirements that ensure the hardware elements contain safety measures to cope with *Random Hardware Failures*. By decreasing the probability of both failure categories, we avoid safety goal violations and accomplish higher integrity levels. Such a complex standard presents many possibilities for researchers aiming to advance state-of-the-art to facilitate compliance to safety requirements.

Furthermore, even though functional safety verification has its particular challenges, it includes several intersections with other areas of interest for developing semiconductors, such as reliability, security, quality, among others. Consequently, the research space that can contribute to compliance with safety standards is vast. In the scope of this work, we have narrowed down the many challenges presented by functional safety compliance to the ones that the EDA industry can support. The EDA industry has a crucial role in the development of integrated circuits. The available tools and methodologies cover the entire product lifecycle range, including support from requirements to post-production tests; the same scenario applies to the development of safety-critical solutions. For that reason, the EDA industry must also undertake the additional challenges for compliance with safety standards.

Even though most semiconductor development activities rely on EDA tools, the final burden of certification is from the companies responsible for the development. Therefore, it is essential to consider the feedback of these companies to understand the actual difficulties and define strategies to facilitate the development of safety systems. Furthermore, during the development of this research, we had the opportunity to contribute with some essential IP providers for the Automotive sector; therefore, we could understand their challenges and narrow down the research topics that would provide the higher contribution. Finally, by understanding state-of-the-art EDA technologies and considering the requests from Automotive IP providers, we have selected the following topics of interest:

- Fault space analysis;
- Early estimation of safety metrics;
- Validation of software tools for compliance with the *Confidence in the use of software tools* clause;
- Enhancements to the functional safety verification process of hardware designs.

1.3. STATE-OF-THE-ART IN FUNCTIONAL SAFETY VERIFICATION

In this section, we will analyze the state-of-the-art on the chosen research topics; for such, we describe the current industry practices and elaborate on the latest findings from the research community.

1.3.1. FAULT SPACE ANALYSIS

This section first defines the faults as seen by ISO 26262 and, after that, briefly describes the commonly known technologies for fault classification, namely Formal Methods, and Fault Injection.

1

FAULT CLASSIFICATION

According to ISO 26262, the safety analysis of a Hardware device must consider the classification of the effect of faults on the circuit's functional behavior; for such, the standard defines all inputs and outputs of the design gates as fault targets. Also, ISO 26262 demands analysis of SA0, SA1, SEU, and SET fault models. Therefore, the fault space to be analyzed englobes the four cited fault models on every input and output of the design gates. For example, Figure 1.9 illustrates a section of a logic design in which red arrows highlight the fault targets.

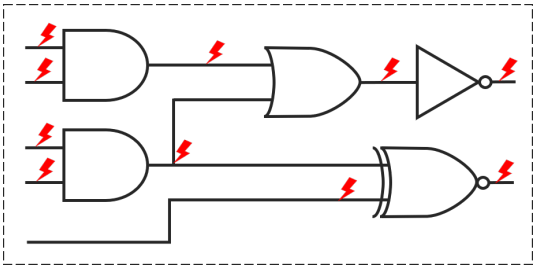

Figure 1.9: Targets for fault analysis

In such an example, the permanent fault space analysis would require the classification of 20 faults, resulting from examining the SA0 and SA1 fault models on the ten fault targets highlighted in the Figure 1.9. The philosophy behind it is to identify the behavior change caused by each fault. Then, based on the circuit's response under the fault's influence, we can classify a suitable fault sub-class. The possible fault sub-classes are:

- Safe faults: these are faults that do not cause any disturbance of safety-critical functionalities; also named untestable or redundant in the DfT community.
- Detected faults: these are faults that can disturb the safety-critical functionalities; hence SMs are deployed to correct them and ensure that they become innocent.
- Dangerous faults: these are faults that can disturb the safety-critical functionalities and are not protected by SMs; therefore, they are an immediate risk for safety goal violations.
- Undetected: these are faults for which the effect is unknown; they can be either Safe, Detected or even Dangerous faults without associated safety mechanisms.

Figure 1.10 illustrates the process of fault classification commonly deployed by the industry for ISO 26262 compliance. It starts with the definition of the fault space; all faults are initially classified as Unknown. Next, engineers deploy Formal Methods for identifying Safe faults; as these are untestable, it is beneficial to remove them before the simulation. Finally, the remaining fault space undergoes FI Simulation; this process classifies the faults as Detected when SM covers them, Dangerous when they affect the design functionalities, and Undetected otherwise. This initial assessment allows the calculation of the Diagnostic Coverage (DC); it represents the efficiency of Safety Mechanisms, and it is essential for ISO 26262 compliance. If the desired DC is achieved, the

process ends. Otherwise, engineers have two possibilities: first, to deploy expert judgment trying to classify the Undetected residual faults, as shown in Figure 1.10; second, to include additional SMs to detect/correct Dangerous faults.

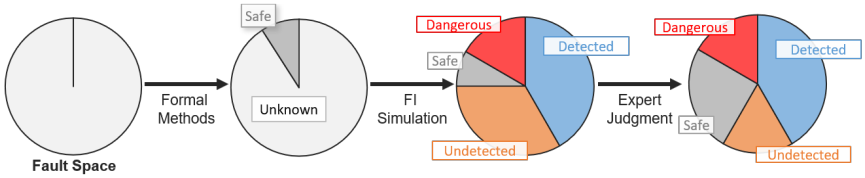

Figure 1.10: Fault Space Analysis

The reduction in the number of Undetected and Dangerous faults improves the DC. As described in 1.2.3, the DC is crucial during the FMEDA for computation of the residual FIT and the safety metrics. The DC is calculated according to the equation 1.5:

$$DC = \frac{Detected}{(Total - Safe)} \tag{1.5}$$

where *Detected* represents the number of faults detected or corrected by SMs, *Total* is the size of the fault space, and *Safe* the number of faults that cannot disturb safety-critical functionalities. The following sections detail the implementation of the lead technologies for fault classification.

FORMAL METHODS

The classification of a fault as Safe is only possible if one can prove that the given fault is untestable; in other words, no existing combination of test stimuli can propagate the fault. Therefore, formal methods appear as a sound alternative since they are not limited to a specific time or state. Instead, the scope is global, and every evaluation context and test stimuli are considered [2]. Consequently, formal methods can exhaustively prove that a fault can never produce any failure; hence, these faults cannot violate safety goals and are considered Safe.

Different EDA vendors explore fault analysis capabilities in their formal solutions. Generally speaking, these solutions automatically generate properties that prove the behavior of the faulty design; for that reason, knowledge of formal languages is not necessary. In addition, they allow integration with FI Simulators providing fault lists optimization and reducing simulation campaigns. Tools used for formal analysis usually apply two main fault analysis techniques, Structural Analysis and Formal Analysis.

1) Structural Analysis: It determines the testability of faults by verifying the physical characteristics of the design. Figure 1.12 illustrates such approach; the figure represents a circuit with combinational logic (g), outputs (out) and fault targets (f). Initially, the user must configure the outputs of the circuit that are relevant for fault propagation, also named Observation Points or Strobes. In the example, as the only Observation Point (strobe) configured for the fault analysis is 'out0', any fault outside of its Cone of Influence cannot propagate to 'out0'; therefore, it is considered Safe. The structural analysis uses three techniques to identify the nature of faults:

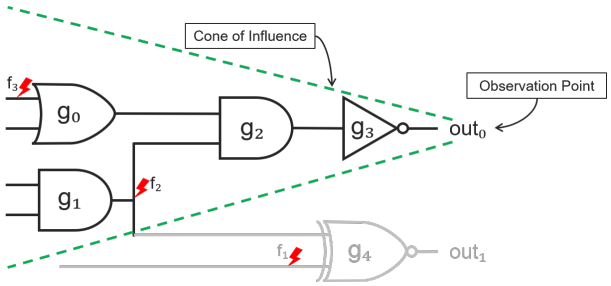

Figure 1.11: Structural Analysis Example.

- Cone of Influence analysis: For the given cone of influence in the figure and considering only one strobe 'out0', a fault 'f1' will never cause a failure; hence it is Safe.

- Activation analysis: The drivers of 'g1' can define the activatability of 'f2'. For example, if 'g1' always outputs the logic value one, 'f2' would not be activatable for SA1 faults. Consequently, an SA1 fault in 'f2' would be Safe.

- Propagation analysis: The combinational logic 'g2' can mask the propagation of a fault in 'f3'. If, for example, 'g2' is an AND gate, with one of the inputs always set with the logic value zero, the effect of a fault in 'f3' would never propagate to 'out0'; hence 'f3' would be Safe for SA1 and SA0 faults.

2) Formal Analysis: Its fundamental theory consists of creating a representation of the boolean function implemented by the design under test, where formal proves can be deployed. Modern Formal tools employ different techniques, and although details of implementation are not disclosed, standard forms of design representation are Binary and Multiway Decision Diagrams [3][4].

To understand the behavioral changes in the faulty design, these tools elaborate two copies of the design representation; the good and the bad machine. First, the tool applies the same inputs to both machines and includes monitors to the observation points; differences in the observation points of both machines indicate the propagation of a fault. Second, by forcing the fault effect in the bad machine, we can determine the fault subclass; this process is repeated for all elements of the fault space. Figure 1.12 illustrates the formal analysis process.

Unlike simulators, formal methods rely on the verification of properties to understand the design behavior. For that reason, such tools automatically generate properties to determine the activation and propagation of faults. The activation analysis indicates whether any combination of inputs can functionally activate a fault. Therefore, the formal engine must verify a property confirming that the fault target can assume a logic value opposite to the fault model, allowing the activation of such fault. Next, the formal engine must prove the properties for every possible combination of input values. If a property is false, the activation of the relevant fault is not possible, and therefore, the

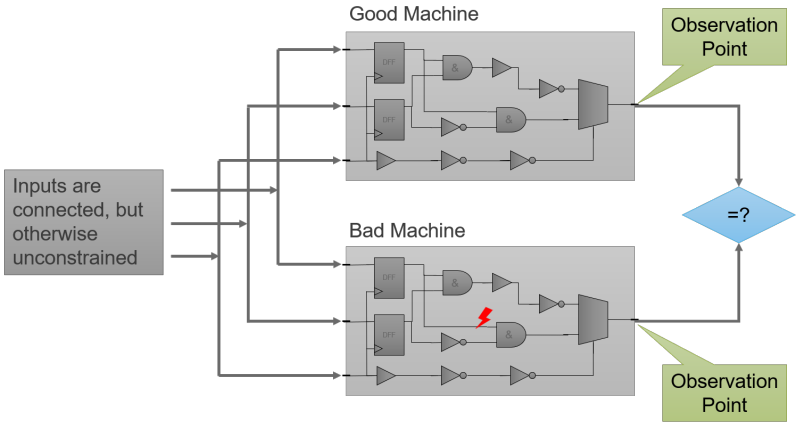

Figure 1.12: Formal Analysis flow

fault is Safe. The example below describes the properties to verify the activation of SA0 and SA1 faults in the fault target *"dut.cpu.loadstore.read_data_enable"*.

- SA0 activation property:
  assert property (dut.cpu.loadstore.read_data_enable == 1)

- SA1 activation property:
  assert property (dut.cpu.loadstore.read_data_enable == 0)

Propagation analysis verifies if there is a combination of inputs that provoke fault propagation to the observation points. Once again, formal properties to perform the analysis are automatically generated and verified concerning all possible input stimuli. However, the properties for the propagation analysis are complex; the formal tool must create a set of assumptions based on the reset conditions, the clock trees, and user inputs; then, it generates properties to describe the propagation paths and monitor the observation points. Finally, for each element of the fault space, the properties are verified by the formal engine; faults that cannot propagate to the observation points are Safe.

The formal analysis capabilities of such tools can also provide additional information about the faults. If, for example, a property that verifies the propagation of a fault is confirmed, the tool can provide the combination of input stimuli that demonstrate the property. Hence, we can retrieve a counter-example for each proven property, showing the necessary conditions for the property to be true. Furthermore, we can replicate such inputs stimuli in the simulation environment, improving the Fault Injection simulation results.

Even in state-of-the-art solutions that demand powerful parallelization capabilities, it is worth noting that formal analysis is still limited due to the state explosion problem. In addition, such engines are resource hungry and cannot find results for all fault space. Therefore, as illustrated in Figure 1.10, the Undetected residual faults are still a problem

requiring expert judgement for additional classification to accomplish sound Diagnostic Coverage figures.

FAULT INJECTION SIMULATION

Fault Injection Simulation is widely used and available in various solutions. It is also the recommended ISO 26262 fault analysis method. Such tools can analyze RTL or gate-level design descriptions and simulate their behavior based on given test inputs. The determination of a faulty design behavior occurs by comparing the outputs of the hardware with and without faults. The flow implemented by Fault Injection Simulators is described below:

1. Elaboration of RTL/GTL design description.
2. *Fault list generation and optimization*: definition of the fault space for simulation. The user can define fault targets, models, and activation times for individual fault campaigns. Also, modern tools include capabilities for pruning the fault space, reducing simulation efforts.
3. *Fault-free simulation*: fault-free behavior of design is simulated for recording the observation points reference values. The observation points, defined by the user, are (1) functional strobes, which store information related to functional outputs, and (2) checker strobes, which indicate how the Safety Mechanisms react.
4. *Fault injection simulation*: For each fault, the faulty design behavior is simulated. The observation points are then compared to the reference values; the differences in the values determine the design behavior under fault.
5. *Fault classification*: If the fault effect is perceived in a checker strobe, then the fault is classified as Detected; if it is noted in a functional strobe, it is classified as Dangerous. Otherwise, the fault is classified as Undetected.

FI Simulation determines the behavior change provoked by a fault when the effect is observable in one of the outputs (strobes). In cases where a fault effect is not visible on the strobes, the tools classify it as Undetected. However, the propagation of faults to the observation points is highly dependent on the stimuli applied to the circuit. For that reason, the classification of a fault as Undetected is not conclusive, as different stimuli could induce fault propagation to the observation points. In these cases, it is common to repeat FI simulation with additional test benches to cause fault propagation. After extensive simulation campaigns are held, the remaining option is to count on expert judgment to analyze the remaining fault space to improve fault classification and, therefore, the DC.

ENHANCEMENTS TO THE FAULT SPACE ANALYSIS

As described above, the industry's state-of-the-art solutions for functional safety verification still leave gaps in the fault space that must be closed for achieving higher integrity levels. As a result, it is common to use Expert Judgment; specialists in the design implementation perform several analyses to identify areas that would not be affected by faults due to design characteristics. Additionally, formal analysis and FI simulation are

a constant source of improvements possibilities as any enhancement in the results or execution times can be crucial. For those reasons, several works explore options for enhancing the Fault Space analysis.

Identifying Safe faults is one of the critical contributions to the flow, as an early classification of these faults reduces simulation efforts and directly improves the DC. In [5], the authors present a method for identifying untestable faults in sequential circuits. For such, it deploys a symbolic simulation to identify flip-flops and circuit lines that cannot be initialized. In [6], the authors propose an alternative method based on theorems that enables a fault injection in any time frame within the unrolled sequential circuit, which implications extend the unobservability propagation of gates to multiple time frames. Furthermore, this technique efficiently identifies conflicting assignments to multiple-node by analyzing logical relationships in the circuit, increasing the identification of untestable faults.

In addition to identifying Safe faults, another essential technique for improving the execution times of FI simulation is optimizing the fault space. One of the most efficient methods is the generation of fault groups by cluster techniques [7]. By grouping hardware components with identical behavior under the influence of faults, we can determine the classification of the entire group by analyzing a single member. Examples of such works are [8], and [9], where authors explore different techniques to group similar semiconductor components into clusters to reduce the number of FI simulations.

Due to high complex characteristics and the number of involved technologies, an ideal Fault Space analysis methodology cannot rely on a single approach. Instead, it should consider a combination of expert design analysis, FI simulations, and formal analysis techniques. An example of such complete methodology is described in [10]. In this work, the authors specify a fault simulation methodology that uses fault propagation probability and clustering approaches for accelerating fault simulation campaigns. Another example is described in [11]. Here, the authors introduce concepts of functional safety analysis; and explain how the state-of-the-art traditional design flow bridges with fault space analysis.

Even though the presented methodologies improve fault analysis and execution time, there is still a gap requiring expert judgment. As part of this process relies on manual analysis by experts, they tend to become expensive, time-consuming, and prone to errors. Consequently, an automated and reliable methodology that decreases manual efforts while fulfilling ISO 26262 requirements is needed. Additionally, a common problem is the lack of representative test cases that capture the complexity of Automotive semiconductors; it is not always obvious to realize how the proposed solutions would escalate in larger designs.

1.3.2. EARLY ESTIMATION OF SAFETY METRICS

As discussed in 1.2.3, the confirmation of the safety requirements defined during the *HARA* is only possible at later stages of the safety lifecycle. Therefore, in cases where achieving safety metrics is impossible, additional iterations through critical development and verification phases are needed. For that reason, a misleading architecture decision before implementation will be exposed only at the final stages of the development, producing a high impact on costs and development time. For that reason, there

is a high demand for techniques that can support engineers with design space exploration of safety features, increasing the confidence in conceptual decisions and avoiding rework. One of the fundamental variables to determine the safety metrics is the DC. Consequently, early-stage evaluations of the DC, together with an estimation of the gate-level design area, enable an accurate prediction of the fault classes and, therefore, the safety metrics. For that reason, several authors explore the correlation of diagnosis in different design abstraction levels; such levels are Virtual Platforms based on Instruction Set Simulators (ISS), RTL, and gate-level.

Methodologies to perform fault injection in Virtual Platforms [12] and also the effects of soft errors are explored in several works as [13], [14], [15], [16], among others. In general, the authors explore the effect of bitflips in the registers of CPUs using Virtual Platforms. These articles include several important contributions: the comparison of faulty behavior in bare-metal software applications and operational systems, the effectiveness of traditional SM for soft errors, software mechanisms for fault detection, among others. However, the fault targets and models in such a high abstraction level are not representative of the gate-level; for that reason, predictions based on these are not accurate.

Trying to improve the accuracy of fault analysis in different abstraction levels, authors start exploring correlation models. In [17][18], the authors propose a multi-abstraction level simulation of embedded processors. The simulation utilizes checkpoints enabling the context switch between the ISS and RTL. The simulation starts with the ISS until the fault activation time. Then, the simulator switches to RTL and inserts a bitflip into the fault target register. Next, the RTL simulation continues until one of the checkpoint registers can capture the fault's impact. Finally, the simulation switches back to the ISS to investigate the fault impact. In [19][20], the authors elaborate a correlation of the fault injection experiments in an RTL microcontroller description with the information available at the ISS. The correlation between RTL and ISS fault injection implies that the probability of an injected fault becoming a failure depends on the set of instructions exercised by the microcontroller. Thus, the likelihood that a given set of instructions triggers a failure is proportional to the processor functional units utilized for each instruction and, consequently, the area of each unit.

The correlation between RTL and gate-level provides a better trade-off between accuracy and effort for predicting safety metrics. Such practice is already explored in other areas of interest. In [21] the authors address test pattern generation in RTL. They propose a fault model and an ATPG algorithm, resulting in excellent fault coverage properties in test patterns, also providing a good estimate of the final gate-level fault coverage. In [22], and [23], the authors propose frameworks for establishing correspondence of gate-level implementation and its reference model specified at RTL. The papers propose techniques to compare the similarities in names, structures, and functions between the gate-level implementation and the RTL counterpart.

The correlation between abstraction levels appears as a good alternative for the prediction of safety metrics. However, as higher abstraction levels represent a subset of the fault space available at gate-level, the estimation errors tend to be significant. For that reason, there is still a lack of methodologies that can support the accurate estimation of safety metrics during the several phases of the hardware development process.

1.3.3. VALIDATION OF SOFTWARE TOOLS

The confidence in the use of software tools tends to be overlooked during the development process. In general, the tool developer is responsible for providing well-tested solutions for their customers. On the other end, the user trusts the tool's outputs are correct and don't need additional verification. However, in the context of safety-critical applications, where a malfunction could cause life-threatening situations, we need to ensure that all possible causes of failures are covered. Therefore, as described in 1.2.4, any software tool that can introduce, or fail to detect, failures in a safety component, should comply with the requirements of the *Confidence in the use of software tools* clause.

The growing adoption of safety standards in the automotive industry results in an increasing uncertainty about software tools. In [24], the authors summarize the tool qualification approach of ISO 26262 and differentiate it from other safety standards. Also, the work describes their first-hand experiences with qualifying development and verification tools. In [25], the author presents a study that ties weaknesses in support environments to software faults. Based on the results, the work provides a guideline for a top-down software toolchain qualification; it also includes a broader view on risks related to tool usage.

In [26], the authors define safety goals for toolchains and suggest a qualification method that takes a systems approach. With this method, software tools are pre-qualified under the assumption that specific properties of the development environment will support the verification of the software tool. In [27], the authors propose using a monitor and fault injection to exercise the functionalities of a software tool. The work defines a process that iteratively activates the tool's functionalities and simulates faults that could impact the given functionality. When all injected faults are detected as expected, we can increase confidence in the measures to avoid malfunctions in the tool.

The state-of-the-art presents attractive alternatives for complying with the requirements for tool qualification. However, the process to prove that a tool's malfunction cannot interfere with a safety-critical functionality presents an additional burden on the tool's developer and users. For that reason, there is a high demand for methodologies that could automate the verification of the software tools, avoiding expensive development processes and additional assurances on the user's side.

1.4. RESEARCH TOPICS

Traditionally, compliance with functional safety requirements is the responsibility of car manufacturers and system providers. However, with the increasing complexity of electronics involved, this burden is now diffusing through the supply chain, namely the EDA companies and design tool providers. Given this scenario and the state-of-the-art previously discussed, innovations in the several challenges of functional safety verification are in high demand. However, on this work's scope, we have focused on the joint hurdles of Automotive IP providers and EDA companies. Next, we describe the research topics and elaborate on issues in each step.

1.4.1. VALIDATION OF SOFTWARE TOOLS

The development of safety components demands compliance with requirements that validate the development environment and tools. As previously discussed, the validation of software tools is mandatory for achieving ISO 26262 certification. Therefore, it is important that aspects of software tool qualification as evaluated in the initial development phases. Failing to fulfill the requirements could make design elements considered unfit for use in a safety-critical context.

1) Requirements for the evaluation of software tools: We first surveyed the demands of safety standards regarding the validation and software tools and then investigated the literature and industry methodologies. Also, the study considered the needs of Automotive IP providers to improve their confidence in the tools used in their development environments. The goal was to establish alternatives to enhance the trust in the software tools without resorting to expensive modifications to their development processes.

2) Methods for the qualification of software tools: The TCL analysis will determine the necessity for Tool Qualification; as previously discussed, such examination is based on the TI and TD parameters. The tool's impact (TI) depends on its role in the safety lifecycle and cannot be modified; however, improvements on TD are possible by enhancing the methods for detecting tool malfunctions. Tool Qualification generally requires demonstration of the tool development process following safety standards; such an approach would be costly for the EDA industry, considering that these tools are constantly evolving and require additional qualification effort for each new release. Therefore, we will address the issue of how to verify software tools' functionalities without modifying the tool development process. Additional verification in the tool outputs enables a better TD, increasing the confidence in the tool's results and avoiding Tool Qualification.

1.4.2. REPRESENTATIVE TEST CASES FOR THE AUTOMOTIVE SECTOR

The availability of representative test cases is an additional challenge when assessing the quality of functional safety verification methodologies. Nowadays, industries do not disclose development lifecycles and verification techniques, and each big player in the automotive sector has its methods and tools. For that reason, it was crucial to define a representative test case to assess the quality of the results proposed in this work. To this end, the following topics are explored in this thesis.

1) Characterization of industry Automotive solutions: Initially, we performed a comprehensive analysis of commercial SoCs from the leading Automotive IP providers; this research leads to the definition of functional blocks present in the various platforms. Also, specifics of the Automotive sector are considered, generating requirements for the proposed solution. This characterization aims to create an SoC definition that is representative of the Automotive industry standards.

2) Evaluation of Open Source platforms: Next, we need to choose a development platform that complies with the previously gathered requirements and is open-source, enabling easy dissemination in the research community. The selection of the CPU, as the central unit of the SoC, considered different processor architectures, performance features (e.g., pipeline stages and memory interfaces), main buses, software stacks, and the possibility of development on multiple abstraction levels (Virtual Platforms, RTL,

and gate-level).

3) Definitions of an Automotive SoC platform: Finally, after defining the requirements to qualify an SoC as representative of the Automotive domain, we can build a solution fulfilling such requirements. The proposed Automotive benchmark comprises all its elements in the format of an SoC, and hence, it was named AutoSoC. The AutoSoC was conceived by analyzing commercial solutions and considering standard development techniques deployed by the industry. The selected architecture considered the availability of software (compilers, debuggers, operating systems, and others) and the feasibility of development in multiple hardware abstraction levels. The suite includes various configurations targeting different levels of safety. Due to its intrinsic characteristics, the AutoSoC qualifies as a perfect test case for assessing the quality of the techniques proposed in this work.

1.4.3. ESTIMATION OF DESIGN SAFETY METRICS

Compliance with safety metrics is a critical step for the validation of hardware designs. As previously discussed, the target values are defined during the *HARA*, on the initial development phases; however, the calculation of the metrics is only possible at later stages of hardware verification. This scenario can be dangerous when the target values are not met, resulting in design modifications and re-work of several lifecycle steps. For that reason, developing methodologies to estimate the safety metrics throughout the lifecycle phases is vital for enhancing the hardware development process. To this end, the research carried out in this thesis focuses on the following topics.

1) Safety metrics calculation for compliance with ISO 26262: Initially, this work conducts a comprehensive examination of the requirements for the computation of safety metrics, considering the guidelines of multiple safety standards. Several safety standards (i.e., IEC 61508) provide methods to support the calculation of safety metrics; therefore, they can offer alternative approaches to compliance with ISO 26262. Furthermore, by considering the available possibilities for the computation of the metrics, we can isolate the design variables required for each method.

2) Correlation of design abstraction level: Next, with the appropriate knowledge of the required design information for calculating the safety metrics, we can understand how to assess this information during the earlier development phases of hardware designs. An important variable to be evaluated is the fault space classification; we need to correlate the behavior of faults in the gate-level to higher design abstraction levels, as RTL and Virtual Platforms.

3) Accurate estimation of safety metrics: Once the correlation approach is obtained, we need to determine a method for estimating the safety metrics at distinct hardware development stages. We have to gather the design information available at different abstraction levels and consider the accuracy compared to the gate-level description—the lower the accuracy, the higher the safety metrics estimation error. By considering these conditions, we can define a methodology for accurately estimating safety metrics at every hardware development stage.

1.4.4. ENHANCEMENTS OF THE FUNCTIONAL SAFETY VERIFICATION METHODS

After validating the support tools, defining representative test cases, and establishing a methodology to estimate safety metrics throughout the development process, the last step is to comply with the functional safety verification requirements. For that reason, we target methods to facilitate compliance with ISO 26262 clauses, as described in the following topics.

1) Functional safety verification requirements: As previously discussed, functional safety verification determines several requirements to guarantee that a failure in a safety-critical system cannot cause life-threatening situations. Such requirements are additional verification steps, comparing with the traditional functional verification of semiconductors. Therefore, in this phase, we aim to understand the specific requirements of ISO 26262 for functional safety verification. In addition to a complete examination of the standard, we had the opportunity to analyze parts of the verification methodologies and primary challenges of some of the essential IP providers in the Automotive industry.

2) EDA tools and technologies: With a better understanding of the requirements, we could address the contribution of EDA technologies to comply with safety standards. The EDA tools and solutions are already critical for the development of semiconductors; this is also becoming a certainty for functional safety verification. In this scenario, we need to, first, learn how the different EDA technologies contribute to the industry verification methodologies, the challenges, and gaps; second, how to close the gaps by improving the tools; and last, how the technologies can support additional steps of the hardware development lifecycle.

3) Enhancing the functional safety verification of Automotive SoCs: This phase aims to determine methods to deploy the EDA technologies to close the gaps of functional safety verification. Every EDA tool has a traditional use that is already extensively deployed. Therefore, even though several contributions can still be made, our focus is on finding alternative employment for the different technologies. For that reason, we can divide this research twofold: first, online fault detection, where EDA solutions contribute by generating hardware mechanisms to enhance fault detection; and second, enhancements to the fault space classification, where we look for alternative deployments of EDA technologies for improving fault classification.

1.5. CONTRIBUTIONS OF THE THESIS

Over the entire course of this Ph.D. project, we are devoted to addressing the research issues at different stages of the safety lifecycle. Our main goal is to propose methodologies to support several development phases and facilitate compliance with ISO 26262. Therefore, Figure 1.13 demonstrates how the main contributions of this thesis support the safety lifecycle as defined by ISO 26262; also, the items below summarize such contributions.

1) Functional Safety Verification Methods and Validation: Initially, we need to ensure that the software tools deployed in the development of safety-critical systems have the necessary levels of confidence; for such, based on the literature review, we propose a methodology for increasing Tool Confidence Level according to ISO26262. Our approach

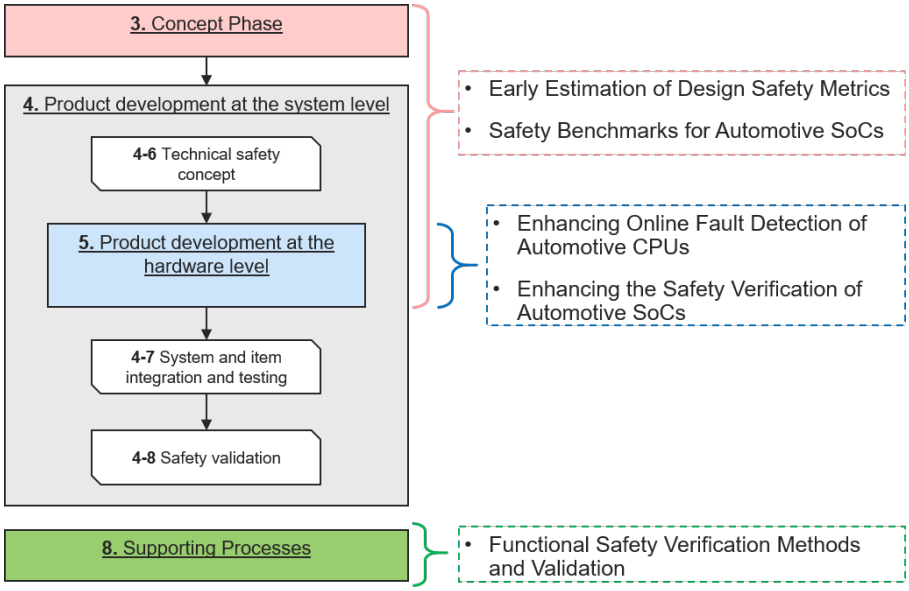

Figure 1.13: Contributions of the thesis to the ISO 26262 safety lifecycle

deploys multiple technologies capable of classifying the behavior of faults; if one of the tools has a malfunction, the classifications will differ, revealing the problem to the user. Furthermore, the validation process is automated by a script responsible for configuring all tools and test environments; at the end, it generates a report comparing the fault classification and highlighting any discrepancies. By providing an automated flow for detecting malfunctions in Fault Analysis tools, we improve the Tool error Detection (TD) capability and avoid the extensive ISO26262 Tool Qualification requirements. Additionally, the report provides further information about the design under test, contributing to engineers with information about the fault space classification.

2) Safety Benchmarks for Automotive SoCs: Next, it was necessary to introduce a hardware design that is representative of the Automotive industry standards. In general, there is limited access to automotive hardware and software solutions; such a scenario becomes a challenge for researchers, who may not verify their work in representative designs or assess the quality of their results. To address these challenges, we propose an open-source industrial-grade benchmark suite. The proposed Automotive benchmark comprises all its elements in the format of an SoC, and hence, it was named AutoSoC. The AutoSoC was conceived by analyzing commercial solutions and considering standard development techniques deployed by the industry. The selected architecture considered the availability of software (compilers, debuggers, operating systems, and others) and the feasibility of development in multiple hardware abstraction levels (Virtual Platform, RTL, and gate-level). The suite includes various configurations with different levels of Safety Mechanisms (SMs), enabling investigation of functional safety aspects. In addition, the AutoSoC demonstrates representative use cases by a set of software ap-

plications, including an Automotive Cruise Control.

3) Early Estimation of Design Safety Metrics: The Safety Metrics verification is only possible at later stages of the development lifecycle, and failing to achieve the required figures demands additional iterations through critical development and verification phases. In other words, it has a high impact on costs and development time. For that reason, this work proposes a systematic approach for the early estimation of safety metrics of Automotive designs. Furthermore, by allowing engineers to estimate fault detection rates before the final development stages, we provide a tool for the design space exploration of safety architectures, improving the confidence in conceptual decisions and decreasing the chances of rework. The methodology is based on the characterization of the design description (RTL and gate-level) and the workload impact concerning fault propagation. Ultimately, the gathered information allows an early assessment of the Fault Injection campaign results at various development phases; and, therefore, it will enable the early estimation of safety metrics.

4) Enhancing Online Fault Detection of Automotive CPUs: The development of Automotive CPUs faces a typical tradeoff between cost and safety. The conventional safety schemes, as Dual-Core LockStep (DCLS), require full redundancy of the hardware area, increasing costs. We propose a methodology that deploys verification EDA technologies to generate Safety Mechanisms (SM), avoiding hardware overhead; as the EDA technologies effectively understand the behavior of faulty designs, we explore such knowledge to assist in generating SMs. First, the process deploys Formal Tools to analyze the propagation of faults in a CPU; for such, we create a set of properties to enable the formal environment to use only pre-selected opcodes for fault propagation. Next, we extract the sequence of opcodes deployed to propagate the fault, also known as the counter-prove of the formal properties. Then, by integrating all opcode sequences, we generate a Software Test Library (STL) capable of detecting faults in the CPU; finally, we deploy FI Simulation to validate the generated mechanisms and assess their quality.

5) Enhancing the Safety Verification of Automotive SoCs: With a sound comprehension of the different stages of functional safety verification; and considering the contributions to previous development phases, we propose a methodology to support the final clauses for hardware safety verification. For such, we introduce an automated approach for identifying the nature of faults overlooked by other technologies, i.e., faults concerning or not concerning safety-critical outputs. For example, suppose the effect of a fault does not affect safety-related functionalities. In that case, there are no Safety Goal violations; hence the fault can be classified as a Safe fault, increasing compliance to safety standards. Additionally, we validate all listed contributions to the safety lifecycle by performing the hardware verification clauses considering the AutoSoC and all proposed methodologies. Such confirmation follows ISO 26262 functional safety requirements, including developing an FMEDA, Failure Rate analysis, and the final metrics calculation.

1.6. THESIS ORGANIZATION

The aforementioned contributions to the ISO 26262 safety lifecyle will be elaborated in detail in the remainder of this thesis, which is organized as follows.

Chapter 2 describes a methodology to validate the software tools deployed for the

functional safety verification of hardware designs. As suggested by ISO 26262, prevention and detection of malfunctions can be accomplished through redundancy in the tasks performed by software tools; we aim to improve Tool Confidence Level (TCL) by detecting errors in the classification of faults using three different technologies. By combining the strengths of Automatic Test Pattern Generators (ATPG), Formal Methods, and Fault Injection Simulators, we can automatically generate a Test Environment that enables the validation of the tools and provides supplementary information about the design behavior.

Chapter 3 narrates the concept and development of the AutoSoC benchmark suite. First, it presents a study regarding the main features available in commercial SoCs from the leading Automotive IP providers; this investigation leads to the definition of functional blocks required for a representative Automotive test case. Next, it describes the determination of open-source platforms as the central unit of the AutoSoC and the development process. The benchmark consists of different safety configurations targeting particular ASILs; each configuration is based on combinations of the available safety mechanisms. Finally, the chapter describes a preliminary functional safety analysis for each benchmark configuration.

Chapter 4 introduces a methodology that can efficiently and precisely estimate the safety metrics of Automotive designs. The technique is based on the characterization of the design to determine how hardware components contribute to fault propagation. Also, by examining the test stimuli applied during simulation, we can rank workloads according to their fault detection coverage. The approach was verified by running fault injection campaigns on distinct gate-level hardware designs, including the AutoSoC. The results demonstrate that the methodology provides an efficient and cost-effective mechanism to support engineers in a confident design space exploration.

Chapter 5 details a methodology for automatically generating Software-Based Self-Test (SBST) for automotive CPUs. For such, we explore the strengths of formal methods in determining optimal test stimuli for fault propagation to create a sequence of commands in the format of a Software Test Library (STL). The approach improves online fault detection during the operational mode of the CPU—the method experiments on the AutoSoC CPU, enhancing the coverage of the digital area of the SoC.

Chapter 6 presents an automated approach to identify and classify faults overlooked by traditional methods. Our methodology recognizes design areas that are not relevant for the safety-critical functionalities of the design; as faults in such parts cannot disturb safety goals, they can be considered Safe during the fault space analysis. The method starts with a comprehensive code coverage analysis to understand the design operational behavior; this behavior is automatically translated into formal properties, enabling the classification of faults by formal tools. Finally, the approach is validated using the AutoSoC benchmark as a test case; we perform the fault space classification, FMEDA, and safety metrics calculation according to ISO 26262 guidelines.

Chapter 7 concludes this thesis and provides an outlook to future research directions.

2

FUNCTIONAL SAFETY VERIFICATION METHODS AND VALIDATION

Safety standards are concerned not only with the validations of the systems under development; but also with the development environment. For instance, ISO 26262 demands a comprehensive validation of the software tools deployed to design and verify such devices. The intention is to avoid that a malfunction could mask or fail to detect failures in the design. For that reason, this chapter proposes a methodology to improve the Tools Confidence Level (TCL) by detecting malfunctions in the tools used for fault analysis. We deploy three different fault classification technologies: Automatic Test Pattern Generators (ATPG), Formal methods, and FI simulators; by comparing the fault annotation from the various technologies, we can identify possible malfunctions. Also, our approach enables the use of test environments generated by ATPG, allowing similar conditions for the fault analysis in the different technologies; and avoiding efforts to develop test benches.

Parts of this chapter have been published in the IEEE 28th Asian Test Symposium (ATS), 2019 [28].

FUNCTIONAL safety verification is one of the most challenging steps for compliance with ISO26262. Particularly for safety-critical applications such as autonomous driving, where failures can cause life-threatening situations. For such applications, engineers must consider every possible source of malfunctions that could lead to impairment in safety-related functionalities. Therefore, in addition to the traditional development flow, ISO 26262 contemplates verification requirements related to the software engines deployed for development and verification. The verification of software tools requires that developers assess the level of confidence in its outputs. If the necessary levels are not demonstrated, the engines must undergo the Tool Qualification process, increasing the complexity and costs of functional safety verification.

In the context of this work, we target the validation of software tools deployed for fault space classification. As previously discussed, the leading technologies for functional safety verification of faults in semiconductors are Fault Injection (FI) simulation and formal methods. FI simulation aims to determine if faults can impact the design outputs and that SMs can detect them. However, as this technology relies on stimuli applied to the circuit, the propagation of faults depends on the simulation testbench (workload). Thus, faults that are not observable must be re-simulated with different stimuli, resulting in complex simulation environments with numerous workloads; if the simulation cannot determine the fault behavior, it must be proven untestable by other means. On the other hand, formal methods are a crucial technology for establishing that a fault is untestable. Its capacity for analyzing design behavior to all possible combinations of test inputs allows the identification of untestable faults and determining test inputs for corner cases.

The tools deployed in the verification of semiconductors are in constant evolution. The nature of such devices requires continuous development of new features to accommodate their demands. For that reason, it is common practice to update the version of the software engines during the lifecycle of a semiconductor. Unfortunately, this scenario increases the difficulty of obtaining Tool Qualification, as such a process increases the complexity of new releases. The Tool Qualification process demand that the development of the software tools comply with safety lifecycles avoiding systematic errors but impacting the time and costs of new releases. Another possibility suggested by ISO 26262 is to verify the correct behavior of a software tool through redundancy in the tasks it performs. Including means for preventing and detecting malfunctions improves Tool Confidence Level (TCL), avoiding the need for Tool Qualification.

This chapter proposes a methodology to verify the correctness of fault classification tools by combining three different technologies, named Automatic Test Pattern Generators (ATPG), FI simulation, and formal methods. For such, we compare the classification given by each technology; discrepancies in the results may expose a malfunction. Additionally, we deploy the test environment generated by ATPG tools for the FI simulation; by doing so, we avoid development efforts and ensure equality between the analysis on both engines. First, we deploy fault analysis by ATPG, generating a verification environment that provides a high fault propagation rate. Then, the FI simulator uses such an environment to perform the classification of the fault space. In parallel, we employ fault analysis by formal methods, identifying untestable faults and determining the behaviors that ATPG does not cover. Finally, the outputs of each tool are verified against each other

to expose malfunctions, increasing the confidence in the tool's results, as required by ISO 26262.

2.1. FAULT ANALYSIS TECHNOLOGIES

This section investigates how different technologies implement fault analysis. The study aims to identify the strengths and weaknesses of each solution and determine how they comply with functional safety requirements. The variation in methods to assess the behavior of faults is crucial to conclude if the technologies can validate each other; we must recognize correlations in the classification given by each tool, allowing the identification of inconsistencies, and therefore, malfunctions.

2.1.1. FORMAL METHODS

As described in 1.3.1, the identification of Safe faults by formal methods is one of the first steps of fault space analysis. As such a sub-class of faults is untestable for any test stimuli, there is no need to undergo FI simulation, optimizing execution times. Formal analysis can also be deployed for the identification of test stimuli that causes fault propagation. Due to their characteristics, these engines must prove properties to every possible combination of design inputs; therefore, they can identify the optimal set of stimuli to support fault propagation. The formal methods will classify each element of the fault space into one of the following fault sub-classes:

- Unpropagatable: There is no combination of inputs that causes fault propagation to observation (strobe) points (Safe faults).
- Detected: At least one combination of inputs causes fault propagation to a checker strobe point.
- Always Detected: The fault always propagates to a checker strobe point, independently of the test inputs.
- Dangerous: At least one combination of inputs causes fault propagation to a functional strobe point.
- Always Dangerous: The fault always propagates to a functional strobe point, independently of the test inputs.
- Unknown: Formal properties could not be proven.

Such tools allow us to retrieve the necessary conditions where a property is verified; such condition is the counter-example. The counter-example demonstrates the test stimuli and circuit conditions used to verify the property. Thus, if we consider the validation of formal tools, we can deploy the counter-examples to demonstrate that the fault classification is correct. Furthermore, the counter-examples for Dangerous faults can also be deployed to confirm the result in FI simulation. Due to these characteristics that allow the verification of fault classification using counter-examples, formal methods are a powerful tool for the validation of fault analysis. Being so, we can deploy formal fault classification to confirm the results of the FI simulation. For such, we need to verify that

the output of the tools does not contradict each other; for example, a fault classified as Unpropagatable by formal cannot be annotated as Detected by FI simulation; this would highlight a malfunction in one of the tools.

It is worth noting that such engines are resource hungry and cannot find results for all fault space due to the state explosion problem. Thus, even in state-of-the-art solutions that demand powerful parallelization capabilities, most properties will be classified as Unknown after a deadline. Therefore, it is common practice to integrate Formal Methods and FI simulation for optimizing the fault space classification; an integrated fault analysis flow allows the identification of Safe faults before the start of the simulation. Thus, the analysis will reduce the number of simulatable elements. Furthermore, after the simulation, we can execute formal methods on the remaining undetected faults to verify if a combination of test inputs would result in fault propagation; then, these can be applied to the FI simulation.

2.1.2. FAULT INJECTION SIMULATION

Fault Injection (FI) simulation is widely used and available in various tools, being the methodology recommended by ISO 26262 for fault space analysis. These tools can analyze RTL or gate-level descriptions of an IC by simulating its faulty behavior. The determination of the fault effect occurs by comparing the behavior of the design with and without faults. Section 1.3.1 describes the flows implemented by FI simulators. The FI simulation will classify each element of the fault space into one of the following fault sub-classes:

- Untestable (Safe) faults
- Detected faults
- Dangerous faults
- Undetected

The simulation of a design provides a comprehensive analysis of its behavior when exercised by a given stimulus. For that reason, by repeating this analysis, including the injection of a fault, we can have definitive evidence of the behavior changes provoked by such a fault. However, if the fault effect is not observable, the analysis is inconclusive; the result could be different when applying different test stimuli. Also, analyzing all fault space for every possible combination of test stimuli is usually impossible in realistic time frames. This situation is a challenge for the verification of safety-critical semiconductors, as discussed throughout this work. However, it also makes it tricky to ensure that the fault analysis results are correct. Furthermore, as the result of the tools (classification of the fault) depends on the simulation workload, the tool verification also must consider this variable. For that reason, a reasonable methodology for comparing the fault classification must contemplate workloads with a higher rate of fault propagation.

2.1.3. AUTOMATIC TEST PATTERN GENERATOR

ATPG engines generate test patterns to identify if an IC contains manufacturing-induced defects. In other words, to distinguish between the correct circuit behavior and the faulty

circuit behavior. When applying the test pattern to the inputs of a circuit, the values observed at the outputs should be monitored. A defect detection happens if any of the outcomes are different from the expected pattern. Nowadays, ATPG is a well-established technology being used in the development of almost all semiconductors. ATPG tools can generate a minimal group of test vectors to achieve acceptable levels of manufacturing defects detection [29][30]. In addition, the engines can create reports about the testability of each defect, allowing the generation of metrics to indicate test quality and test execution time. The ATPG will generate a fault report classifying each element of the fault space into one of the following sub-classes:

- Untested: The ATPG engine could not process the fault.
- Ignored: The current scan chain configuration cannot reach the fault.
- Tested: The ATPG can test the fault.
- Redundant: The fault has a redundant behavior compared to other Tested faults; therefore, it is also Tested.
- Untestable: The ATPG engine could not test the fault.

Usually, an ATPG flow receives a gate-level description and specification of the scan chains as inputs. Then, it verifies if the implemented scan chains can ensure the required levels of testability. If affirmative, it generates a fault model and test patterns to assure propagation of fault effects to the design outputs. Typically, the test patterns and expected results are programmed in a Test Equipment (TE) used in semiconductors manufacturing tests. The TE applies the test patterns in the circuit's inputs and monitors the outputs to verify if the values are the expected ones. We propose a similar approach using FI simulation. However, instead of using TE, we apply the test patterns on the design simulation and employ the strobe functionality to monitor the outputs of the design. During the Fault-free simulation, the simulator stores the strobe values, defining the expected output pattern. Afterward, the simulation of each fault is executed using the same inputs and monitoring the outputs. By following this approach, we can use the propagation capabilities of ATPG to identify behavioral changes caused by injected faults.

The use of the ATPG is an established practice in the DfT domain. After manufacturing, it is critical that semiconductors are tested to ensure the ICs are functioning as expected; such tests should cover as many components as possible. For that reason, ATPG tends to have very high coverage rates, propagating most of the faults to the scan chain outputs. The fault propagation potential of ATPG test environments is a decisive benefit for compliance with functional safety. However, ATPG focuses on manufacturing tests, where the target IC is in test mode. In the case of functional safety verification, we must demonstrate that the semiconductor is capable of coping with *Random Hardware Faults* while keeping its operation. Nevertheless, the ATPG test environment can still support the validation of software tools.

Table 2.1 summarizes the strengths and weakness of each technology. Considering this examination, we propose a methodology that highlights the strengths of FI simulation, formal methods, and ATPG to validate functional safety verification tools.

Table 2.1: Fault Analysis Technologies Comparison

Technology	Strengths	Weaknesses
FI Simulation	- Comprehensive behavior analysis - Recommended by ISO26262	- Single test input at a time - Multiple simulations to propagate all faults - High Testbench development efforts
Formal Methods	- Analysis of all possible test inputs - Analysis of untestable faults - Generates test inputs for corner cases	- Time-consuming - Not able to determine behavior of all faults
ATPG	- Automatically generated Testbenches - High fault propagation rate	- Focus on manufacturing tests - No analysis of untestable - Do not reach corner cases

2.2. SOFTWARE TOOLS VALIDATION METHODOLOGY

This section describes the application of three fault analysis technologies for software tools validation according to ISO 26262. The methodology highlights the strengths of FI simulation, formal methods, and ATPG to generate a comprehensive fault analysis report. The execution of such tools is controlled by an application that automates the fault analysis flow and validates the results by comparing them. The Fault Checker application implements a control flow that allows the configuration of multiple software tools representing each technology. For each technology, the application parses the fault classification reports and verifies discrepancies in the fault annotations; all findings are synthesized in a final report allowing the identification of malfunctions in the software tools. In the following sections, we describe the steps implemented by the Fault Checker application, named Configuration, Execution, and Report.

2.2.1. CONFIGURATION

The concept behind the proposed methodology is to develop a generic flow allowing the deployment of verification engines from multiple vendors. For that reason, before the execution, the users must provide the required configuration for each tool. Thus, the flows start with selecting one representative of each analysis technology; an ATPG engine, an FI simulator, and a formal solution. In addition, the chosen tools must include fault analysis capabilities, and the naming convention for fault classification must be known.

Initially, the user must provide scripts to control the execution of the analysis flow on each tool; even though all modern tools provide graphical user interfaces, these engines are controlled by scripts that allow automation of design verification in production. Next, the user must provide a set of rules that enable mapping the fault classification from each tool. For example, the Fault Checker entails knowing that a fault classified as Tested by ATPG is equivalent to the Dangerous annotation by FI simulation; these rules will establish the creation of warnings during the report phase. Finally, the user also must provide the design information required for the execution of each technology, i.e., design description and library files, scan chain configuration, strobes, fault targets, and fault models. Figure 2.1 illustrates the configuration phase of the Fault Checker application.

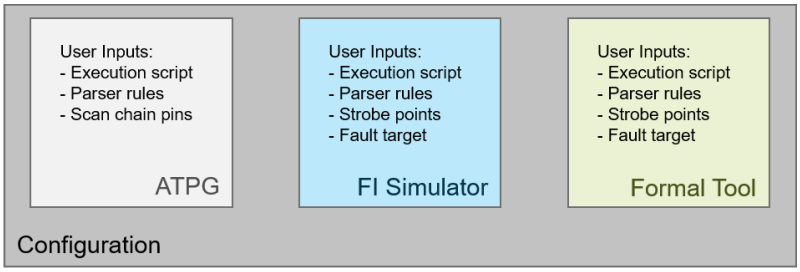

Figure 2.1: Fault Checker Configuration Phase

2.2.2. EXECUTION

After the configuration of all pre-requisites, the Fault Checker can start the execution of the fault analysis flows. The control of the fault classification from each technology is based on the execution scripts provided during the configuration phase. Furthermore, aiming to improve performance, we split the execution into two threads. First, the *Execution Thread 1* initiates with the ATPG flow, performing the fault classification and generating the simulation environment; next, the ATPG testbench and test vectors are configured in the FI simulator, allowing the analysis of the fault space in the same conditions of the ATPG flow. Finally, the *Execution Thread 2* is responsible for controlling the Formal Tool flow; as such engine does not require a test environment, its execution is independent and can be dispatched in a separate thread. Figure 2.2 illustrates the execution phase of the Fault Checker application.

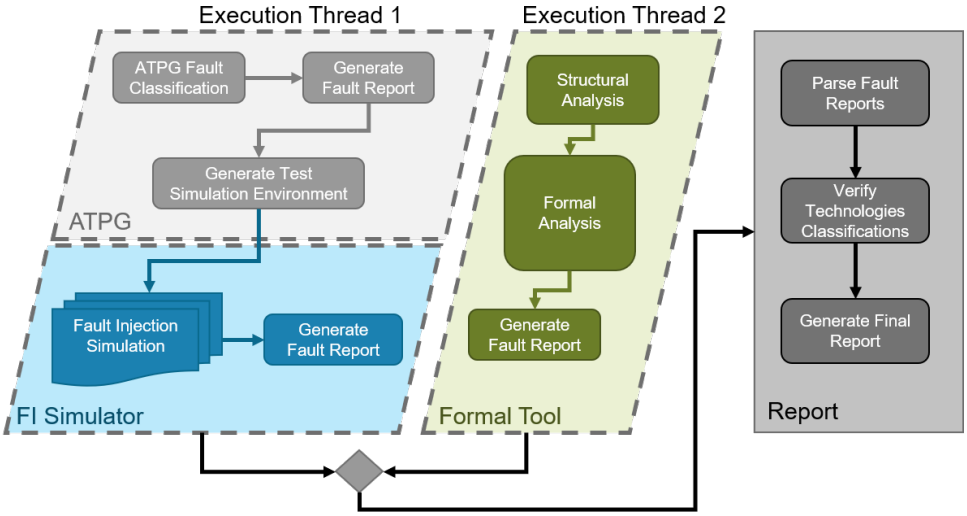

Figure 2.2: Fault Checker Execution and Report Phases

The ATPG flow initiates with the configuration of the test and fault models. The available options may vary on engines from different vendors, but the overall flow is similar.

We initially execute commands for building, testing, and verifying the test model; these will configure the parameters to generate test patterns. Then, we select cell boundary as the fault model; this option shows similar fault patterns compared to the requirements of ISO 26262. Next, we execute the command to generate static patterns targeting the test of the circuit logic and the scan chain logic. Additionally, the options for compacting the test patterns are enabled; this will increase the execution time of the ATPG flow; however, the smaller test vectors demand less effort during the FI simulation, causing an overall improvement in execution time. Finally, the ATPG engine must produce a simulation environment to reproduce the fault analysis with the generated test patterns; such an environment is composed of a Verilog testbench and test vectors, allowing the verification of the ATPG test in simulation. At the end of the ATPG flow, the tool generates a fault report, including the annotation given to each fault in the fault space.

The FI simulation will follow the same flow described in 1.3.1. The main difference is that the simulation will deploy the simulation environment generated by ATPG. Hence, the FI simulator can reproduce the test stimuli and conditions used during the ATPG analysis. For that reason, we can expect that the FI simulation fault classification is equivalent to the ATP classification, allowing the validation of the results from both tools. Furthermore, as the FI simulator must wait for the conclusion of the ATPG, we opt to serialize the execution of both flows, requiring only a single CPU for the control task.

On the other hand, the Formal tool is independent of the separate flows. For that reason, we can administer it in a separate thread parallel to the execution of the ATPG and FI simulation. The flow executed by the Formal tool is described in 1.3.1. The addition of a third technology aims to complement the validation achieved with the previously described flows. Formal methods are a robust tool for identifying Safe faults; the Formal definition of Safe is compatible with the description of Untestable from FI simulators; therefore, the addition of Formal close a gap not covered by ATPG. Also, by including a third fault analysis technique, we improve the chances of identifying the source of malfunctions. In case of incompatible annotations between the tools, we can use a voting strategy to define which tool has a malfunction.

2.2.3. REPORT

After the finish of the execution of both threads, the Fault Checker initiates the Report phase. Initially, the application retrieves the fault report from each tool and parses the results based on the parser rules introduced during the Configuration phase. The parser rules must provide an association between the fault annotations from the different tools. The Fault Checker must be able to recognize discrepancies between the annotations to alert possible malfunctions. Next, the application generates a detailed report in CSV format; the report lists every fault space element, including the fault classification of each tool and the result of the Fault Checker analysis. Table 2.2 illustrates a sample of the Fault Checker report. The parameters highlighted in the report are described below:

- Fault Target: The hardware component where the fault was injected for the analysis;
- Fault Model: The model selected for representing the behavior of the fault;

- Formal Classification: The result of the Formal analysis regarding the circuit behavior changes caused by a fault;
- FI Simulation Classification: The result of the FI Simulation regarding the circuit behavior changes caused by a fault;
- ATPG Classification: The result of the ATPG analysis regarding the circuit behavior changes caused by a fault;
- Checker Result: The result of the Fault Checker application analysis regarding the association of the fault annotation on each technology.

The result of the Fault Checker application analysis will be a *Pass* or a *Warning*. A *Pass* indicates that all the fault annotations for a given fault are compatible according to the parser rules; if there are no discrepancies between the technologies, we can assume there are no malfunctions. A *Warning* will indicate that the user must verify the results for a given fault, as it may represent an error caused by a malfunction in one of the tools. For example, if a given fault is classified as Detected by FI simulation and as Safe by Formal, the discrepancy in the annotation indicates a possible malfunction in one of the tools.

Table 2.2: Fault Checker Report Example

Fault Target	Fault Type	Formal Classification	FI Simulation Classification	ATPG Classification	Checker Results
dut.u0.rst	SA0	Dangerous	Detected	Tested	PASS
dut.u0.sig1	SA1	Safe	Undetected	Ignored	WARNING
dut.u0.sig2	SA0	Dangerous	Undetected	Ignored	WARNING
dut.u0.sig3	SA1	Dangerous	Detected	Tested	PASS
dut.u0.iNsT0.0	SA1	not_listed	not_listed	Tested	WARNING

In addition to malfunction evidence, the report provides supplementary information to understand the behavior of a faulty design. For example, signal "dut.u0.sig2" in Table 2.2, is annotated as Undetected by the Simulator and Ignored by ATPG. Still, the fault is listed as Dangerous by Formal, meaning that formal analysis identified at least one test stimulus that can propagate the fault to a strobe; FI Simulation can use such information to achieve detection of this fault. Another example to highlight is "dut.u0.sig1", where Formal classified the fault as Safe, while the other tools annotated it as Undetected and Ignored. In this case, results from the formal analysis can demonstrate that the fault cannot propagate to a strobe, and therefore can be considered untestable. The report will highlight any other discrepancy between the fault analysis with a *Warning*; as illustrated by signal "dut.u0.INsT0.0".

2.3. EXPERIMENTS AND RESULTS

This section describes the validation process of the proposed methodology. First, we describe the adopted setup, the configuration of the tools, and the target designs. Then, we demonstrate our results and explain the benefits and limitations of the Fault Checker

application. The following validation aspects were considered: Detection of malfunction in the tools via detailed report; Application of fault analysis results to support functional safety verification of the design.

2.3.1. EXPERIMENTS SETUP

The verification of the methodology deploys the Fault Checker application on designs with different characteristics. First, the Fault Checker is configured with the representatives of each technology to execute the fault analysis. Our work has adopted Cadence® Xcelium™ Fault Simulator (XFS), Cadence® JasperGold (JG) Formal Verification Platform Functional Safety Verification (FSV) and Cadence® Modus DFT Software Solution ATPG component, as the software engines for validation.

The selection of the designs contemplated different levels of complexity and the availability of functional testbenches for assessing the quality of our solution. The measurement of complexity considers the size of the fault space in each circuit. As previously discussed, ISO 26262 defines all cell ports in the circuit's gate-level as fault targets. The target circuits are available on the IWLS 2005 benchmark [31]. The designs are:

1. Up-Down Counter: 4 bits adder containing 81 cell ports;
2. Memories: Two memories with CRC, containing 1391 cell ports;
3. AC97: An Audio Codec Controller compatible with Wishbone bus, containing 28610 cell ports; and
4. Conmax: An interconnect matrix IP core featuring parameterized priority-based arbiter, with 76727 cell ports.

The circuits were synthesized using the standard cell reference libraries provided with Cadence 45nm Generic Process Design Kit (GPDK) [32].

Initially, we deployed Designs (1) and (2) to verify that the Fault Checker application worked properly. As the designs are smaller, it was possible to manually check the classification of each fault to ensure the correctness of the final report. Next, we verified the behavior of the Fault Checker application when analyzing larger designs by examining the other circuits. In addition, for circuits (3) and (4), we compared the results with FI simulation results using the functional testbenches only; we aim to ensure that the fault propagation levels are high and cover a large percent of the fault classified by the FI simulator. The following sections describe our results.

The experimental setup consists of two Intel Xeon E5-2680 CPUs with 16 Cores and 252 GB of memory each; the Formal flow executed on CPU1 and ATPG followed by Simulation Flow in CPU2. In addition, the FI simulation flow deploys parallelization for several fault injection jobs; by doing so, we improve the overall execution time of the flow.

2.3.2. RESULTS

To validate the Fault Checker application, we have performed the described flow using the previously defined target circuits. Table 2.3 illustrates the results. The table details the size of the fault space, the rate of fault detection from the FI simulation, and the number of *Pass/Warning* indications resulting from the Fault Checker analysis.

Table 2.3: Fault Checker Results

Design	Faults (SA0/SA1)	Detection Rate	PASS	WARNING
Up-Down Counter	162	100%	162	0
Memories	2782	99,78%	2776	6
AC97	57226	99,77%	57108	118
Conmax	153454	99,80%	153191	263

For the Up-Down Counter circuit analysis, the Fault Checker confirmed that all faults have equivalent classifications. As the example is relatively simple, the different technologies can determine that all faults propagate to observation points (strobes); and have compatible annotations.

In the analysis of the Memories design, the application detected six faults with discrepant classifications. In this example, the *Warnings* were due to annotation of Safe Faults by Formal and Undetected by the Simulator. For these faults, the Formal analysis proves that the faults are untestable and can be disregarded.

On the AC97 design, the Fault Checker was able to detect 118 faults with distinctive classifications. From these, 49 faults were classified as Safe by Formal and Undetected by the Simulator and declared untestable. Additionally, 23 faults were annotated as Dangerous by Formal and Undetected by the Simulator; FI simulation may detect these faults by applying the counter-examples from Formal. Other 46 faults were considered Undetected by FI simulation and ATPG, and Unknown by Formal, indicating that none of the tools could define the possible behavior of these faults, and they require manual analysis. Finally, six faults were in cell ports related to power that are not relevant for functional safety verification.

During the analysis of the Conmax design, the methodology detected 263 discrepancies between the tools. From these, seven faults were classified as Dangerous by Formal and Undetected by FI simulation. Meaning that results from Formal can be applied for detecting these faults during simulation. The other 256 faults were classified as Redundant by ATPG, Undetected by Simulation, and Unknown by Formal. However, as the classifications are not conclusive, the user must analyze these faults manually.

To analyze the capability of the methodology for fault classification, we compared the Fault Checker results with results from fault injection using functional testbenches. It is important to remember that the validation of the FI simulator should deploy an environment it a high detection rate; as Undetected is a weak annotation that could change with different test stimuli, it may mask malfunctions.

The AC97 and Conmax designs include simulation environments for verification of their functionalities. By deploying such an environment, we can compare the detection rates of a functional verification environment against the ATPG simulation environment. Figure 2.3 illustrates the comparison of the FI simulation results when deploying the functional testbenches and the Fault Checker; the additional detection provides a comprehensive environment for validation of the FI simulation results.

Table 2.4 details the results of the FI simulation of the AC97 and the Conmax designs. Due to the characteristics of fault propagation provided by the ATPG Testbenches,

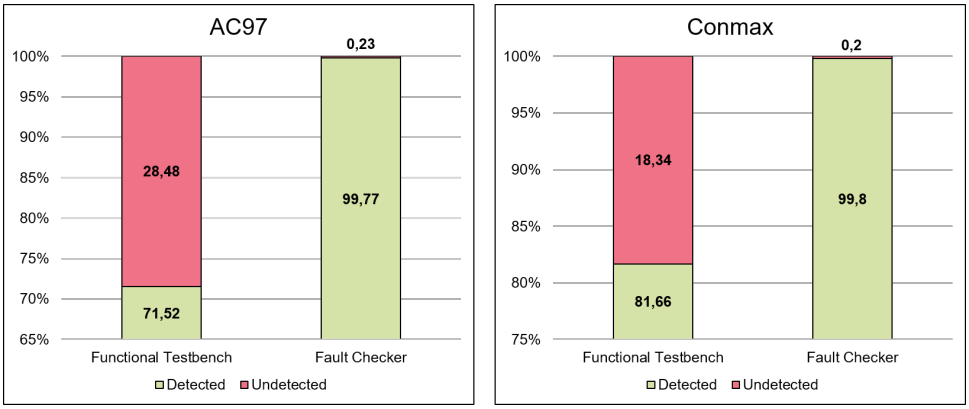

Figure 2.3: Fault Detection Comparison

after one execution of the Fault Injection campaign, the Fault Checker achieves a fault Detection Rate improvement of 28,2% for the AC97 and 18,2% for the Conmax. Hence, these figures represent the number of FI simulation results where malfunctions could be masked, compromising the confidence in the use of the tool.

Table 2.4: Fault Detection Comparison

Design	Faults (SA0/SA1)	Functional Testbench		Fault Checker	
		Detected	Undetected	Detected	Undetected
AC97	57220	71,50%	28,48%	99,77%	0,21%
Conmax	153454	81,66%	18,34%	99,80%	0,20%

The Undetected classification is inconclusive for fault analysis. Undetected faults must be proven Untestable to collaborate to ISO 26262 metrics and are more likely to mask a malfunction in a tool. For these reasons, we want to achieve as many detected faults as possible. To achieve the same level of fault detection with functional testbenches, we would need to repeat the FI campaign with new test inputs until detection rates are acceptable. However, such a scenario demands high efforts in developing test environments and even longer FI simulation execution times.

2.3.3. DISCUSSION

The results demonstrated above corroborate with the selected evaluation criteria. First, the deployment of multiple fault analysis technologies enables the detection of erroneous fault classifications. The proposed methodology allows high confidence in tool error detection, resulting in a sound Tool Confidence Level (TCL). A software tool with TCL1, as described in 1.2.4, doesn't require Tool Qualification, avoiding significant efforts on documentation and analysis for compliance with ISO 26262 [33]. Second, identification of Safe faults collaborates with ISO 26262 compliance. By proving that a fault is untestable, we can disregard it, decreasing the total number of faults to be simulated and

improving ISO 26262 metrics [34]. Third, the proposed methodology achieved substantial fault detection rates; using ATPG test vectors during the simulation and identifying Dangerous faults by Formal provides extra information about the design behavior. In summary, we can apply the proposed methodology to support the following aspects of ISO 26262 functional safety verification:

- Avoid efforts with Tool Qualification by automating tool error detection.
- Identification of Untestable Faults allows improvement of ISO 26262 metrics and reduction of the number of faults to be simulated.
- Fault supplementary data supports additional information for the fault injection campaigns.

Even though we have achieved high fault detection rates, we must consider that the examples used were of average complexity. If the methodology is applied to validate the software tools in the lifecycle of an industry-grade semiconductor, designers would have to decide how to deploy the Fault Checker. As the design under development may not be available, designers could start with example designs; pre-validating the tools and development environment. Later, they may repeat the execution with the actual design, leveraging the additional design information before the final clauses of hardware verification. Another aspect to acknowledge is the possibility of changes in the fault propagation patterns when ATPG scan chains are disabled. For that reason, designers should consider this effect and employ formal results to assess differences in the classification of the faults.

2.4. CONCLUSIONS

Due to the harsh requirements for *Random Hardware Failures* tolerance, functional safety verification is challenging for ISO 26262 compliance. As part of this process, fault space analysis becomes an extensive procedure that is usually repeated numerous times until the metrics for fault detection are achieved. Furthermore, ISO 26262 requires specific criteria to determine the level of confidence in the adopted software tool, increasing the efforts even further.

As the initial contribution of this thesis, we propose a methodology that deploys ATPG and Formal to support the validation of FI simulation and decrease the overall efforts of ISO 26262 compliance. The Fault Checker application enables the use of test environments created with ATPG tools for the simulation of faults; also, it integrates the use of Formal to identify untestable faults. Formal results allow the optimization of the fault space, reducing the number of faults to be simulated, and the generation of test vectors to detect corner cases.

Additionally, the application generates a fault report to identify potential software tools malfunctions. The inclusion of redundancy as a method to detect malfunctions in tools is a suggested method for achieving ISO 26262 Tool Confidence [33]. The validation of the development environment and software tools is a prerequisite for compliance with the safety life cycle requirements.

3

SAFETY BENCHMARKS FOR AUTOMOTIVE SOCS

This chapter addresses the constraints posed by the lack of representative open-source automotive designs. Furthermore, the limited access to such designs and industrial methodologies challenges the research community, making it difficult to assess the quality of their results and compare distinct methodologies and results. For that reason, we propose the AutoSoC, an automotive SoC benchmark suite that includes hardware and software elements and is entirely open-source. The objective is to provide researchers with an industrial-grade automotive SoC that includes all essential components, is fully customizable, and enables analysis of functional safety solutions and automotive SoC configurations. Additionally, we describe the benchmark's available configurations, including an initial assessment for ASIL B to D configurations.

Parts of this chapter have been published in the IEEE 38th VLSI Test Symposium (VTS), 2020 [35].

In recent years, technological advances enabled the employment of automated systems to control driving tasks. The idea of electronic devices having complete control over a vehicle promises to change the concept of mobility soon. However, allowing computers to handle all the tasks in a car requires high complexity systems and significant safety concerns. The development of autonomous vehicles applications, where a system failure could cause life-threatening situations, entails state-of-the-art challenges on different aspects of system development. Therefore, concerns with Reliability, Security, Quality, and compliance to Safety Standards are high priority. This scenario requires adopting new techniques and methodologies that will facilitate the development and verification of these applications.

Several organizations are working to close the technological gap for autonomous vehicles. However, to assess the quality of their solutions, it is necessary to compare the results against industry standards. Nowadays, development life-cycles and verification techniques applied by industry are not disclosed, and each big player in the automotive sector has its methodologies and tools. In addition, there is limited access to automotive hardware and software solutions. This scenario is a challenge for researchers, who may not verify their work in representative designs or assess the quality of their results. For that reason, there is a high demand for a suite of open-source benchmarks that would enable research on the different aspects of Automotive applications development.

As part of the efforts for developing solutions to address the demands of autonomous driving, industry and academia are investing in research on several related areas. Several works are exploring aspects of fault-tolerance in hardware architectures [36][37], software design [38], operational systems [13], among others. In [39] the authors provide a broader look at specific reliability challenges for autonomous systems, for both automotive and robotics. Several works are also discussing the security issues imposed by these applications [40]; including challenges with hardware attacks [41], and secure in-vehicle communication [42]. Although several works include significant contributions to advance the state-of-the-art, they all have some common pitfalls. First, experiments are usually not performed on representative designs. Results may be compromised by a lack of comprehensive test cases, which should be based on representative SoCs; these must include operating systems and software applications typical of their domain. Also, such systems should be fully open-source, allowing different researchers to assess the quality of the results by comparison. Even though some components of such systems are available in the community, to the best of our knowledge, no open-source package including SoC hardware models, OS, and SW applications that is representative of the Automotive sector is available.

This chapter presents an open-source industrial-grade benchmark suite, aiming to address challenges previously discussed. The proposed Automotive benchmark comprises all its elements in the format of an SoC, and hence, it was named AutoSoC. We formulate the AutoSoC by analyzing commercial solutions and considering standard development techniques deployed by the industry. The selected architecture considered the availability of software (compilers, debuggers, operating systems, and others) and the feasibility of development in multiple hardware abstraction levels (Virtual Platform, RTL, and gate-level). The suite includes numerous configurations with different levels of Safety Mechanisms (SMs), enabling investigation of functional safety aspects. There-

fore, the AutoSoC appears as an exciting candidate to support Automotive research. The main contributions of our work are:

- Launch the initiative for an open-source SoC benchmark suite for Automotive applications
- Provide a solution for integrating inter-layer components and their interoperability required for an automotive SoC development
- Demonstrate representative use cases by a set of software applications including an automotive cruise control
- Validate the concept by including a preliminary safety assessment targeting different ASIL configurations.

The AutoSoC benchmark suite is available for download in *http://www.autosoc.org*.

3.1. SAFETY STANDARDIZATION AND BENCHMARKING

Nowadays, the implementation of highly automated safety-critical systems demands complex integrated circuits. These applications are composed of many HW elements, executing an equally extensive collection of SW elements, often from third parties. This complexity has created a strong demand for semiconductor standardization initiatives to guarantee uniformity, interoperability, and repeatability of the many activities required by a safety lifecycle. As previously discussed, safety standards as ISO 26262 present guidelines for demonstrating the levels of safety of automotive designs. As such, there are a variety of possibilities for compliance with the safety guidelines.

That vastity of options is a challenge from several points of view. For example, despite the ISO 26262 providing a mathematical approach to quantify the probability of *Random Hardware Failures*, it is very effort-intensive to apply it and quickly compare the effectiveness of each proposed solution. The results are highly dependent on the chip architecture and the related SW application. The same challenge exists for the safety verification activities (e.g., fault space analysis) required to confirm some functional safety properties' effectiveness, such as the diagnostic coverage. The time spent setting up each fault injection campaign for each different architecture solution makes it unpractical to use it during the exploration phase, limiting the creativity and design space exploration. Another challenge is caused by the interaction between several different properties and requirements. For example, a typical approach to achieve high diagnostic coverage is the so-called loosely coupled lock-step; i.e., the same SW is executed redundantly in two different processing cores and compared by a third element. The resulting diagnostic coverage highly depends on the execution of the redundant SW (e.g., if it is a task per task or instruction per instruction redundancy, if the OS is in common or shared, etc.); also, it depends on how often the two SW executions are compared. Additionally, it is necessary to evaluate the so-called Diagnostic Time Interval, i.e., how often it is possible to perform that comparison and the time required by the SM to compare and detect the potential failure. Furthermore, it is necessary to evaluate the degradation of performance (e.g., in terms of worst-case execution time) that the lock-step comparisons produce to the data traffic of the nominal functionality.

The complexity described by the previous examples indicates the vital need for an open-source benchmarking environment, providing scientists with a ready-to-use and clearly defined platform to implement and test safety solutions in a comparable way. That platform, for example, should allow researchers to compare two different implementations of the loosely coupled lock-step scheme. Another use case for that benchmarking environment is the measure of the application overhead caused by the execution of Software Test Libraries (STLs). Once more, the availability of a standard benchmark will allow a transparent and well-defined comparison of the impact to the application caused by two different STL implementations.

3.2. AUTOMOTIVE SOC ARCHITECTURES

This section describes the analysis of commercial automotive SoCs that led to the definition of the functional blocks of the AutoSoC. The gathering of requirements for the proposed SoC considered the main features available in well-known automotive solutions. The objective of this characterization was to create an SoC that is representative of the industry standards.

3.2.1. INDUSTRY SOLUTIONS CHARACTERIZATION

The industry is embedding several features in SoCs targeting different in-vehicle applications. The so-called automotive ecosystem includes solutions for infotainment, powertrains, network communication, and automatization of driving tasks. All those features require robust solutions that must consider aspects of functional safety and security. Although different commercial solutions are available, architectures generally have similarities that can be explored to define a set of requirements for an Automotive SoC. The gathering of requirements for the AutoSoC was based on an analysis of the datasheets of commercial Automotive SoCs. We considered the main characteristics of available solutions to identify common aspects that the industry considers mandatory. In general, this investigation considers the following domains:

1. Hardware Architecture: common architecture characteristics;

2. Safety: what components of the SoCs are considered for functional safety compliance and which safety mechanisms are usually implemented;

3. Security: which security features are available;

4. Other: commonly available peripherals (e.g. communication protocols, GPUs, Audio/Video DSPs).

One notable common characteristic among the evaluated solutions is the availability of multiple CPUs. In general, dedicated hardware components are available for safety-critical and application-specific operation. This concept allows the deployment of powerful CPUs for applications with high processing demands (e.g., video processing), while safety-critical applications are executed in CPUs with dedicated safety mechanisms. For example, the Renesas R-Car M3 [43] includes two CPUs for common applications and

an additional Dual-Core LockStep (DCLS) CPU for safety-critical applications. The Infineon AURIX [44] and Texas Instruments TDA2SG [45], follow a similar concept by including a CPU and separated cores for dedicated functionalities. DCLS is the most common safety mechanism available for CPUs. Industrial solutions usually deploy Error Correction Codes (ECCs) and Parity for the memories, including RAMs and caches. DCLS, ECCs, and Parity have an advantage regarding functional safety analysis. These SMs are introduced by the recommendations of ISO 26262 [34] and include a reference of their fault coverage capabilities. Hence, by deploying any of these SMs as described in ISO 26262, the referenced DC can be directly used during functional safety analysis.

The other components in the analyzed SoCs are related to communication protocols, application-specific, security, and infrastructure peripherals. The commercial solutions generally implement various communication peripherals, including automotive protocols as CAN and FlexRay; also, general protocols as Ethernet, SPI, and I2C. Another common characteristic is the availability of video and audio dedicated hardware. As most of the SoCs aim to Advanced Driver-Assistance Systems (ADAS) applications, they include peripherals like GPUs, video codecs, Image Processing Units, and audio DSPs.

Apart from proprietary features that are not detailed in the security domain, the most common components are cryptography engines, like Advanced Encryption Standard (AES), Data Encryption Standard (DES), and Hash. Also, some solutions provide access control features like firewalls and protected memory areas. Additionally, every analyzed solution included infrastructure peripherals like JTAG, UART, GPIO, and debug components.

The set of characteristics gathered during the investigation points to conventional features that describe the automotive industry requirements. Therefore, the extension of safety-related components, application-specific units, automotive protocols, and security cores can be established as the basic set of features for a representative Automotive SoC. The summary of common characteristics observed in the evaluation of commercial solutions is detailed in Table 3.1.

Table 3.1: Summary of Commercial SoC Analysis

	Renesas R-Car M3	**Infineon AURIX**	**Texas TDA**
Safety CPU with DCLS	+	+	-
Memories with ECC	+	+	+
Second CPU (no SM)	+	+	+
Dedicated Video IPs	+	+	+
Automotive Peripherals	+	+	+
Security Cripto IPs	+	-	+

3.2.2. AUTOSOC FUNCTIONAL BLOCKS

Based on the characterization of industrial solutions, we establish an initial architecture of AutoSoC. Initially, we define functional blocks to cover the minimum features required for a representative automotive benchmark suite. Such a concept is also essential to keep the design modular. As a result, the various versions of AutoSoC can deploy distinct

hardware components to cover the requirements of each functional block. Figure 3.1 illustrates the outcome of our analysis.

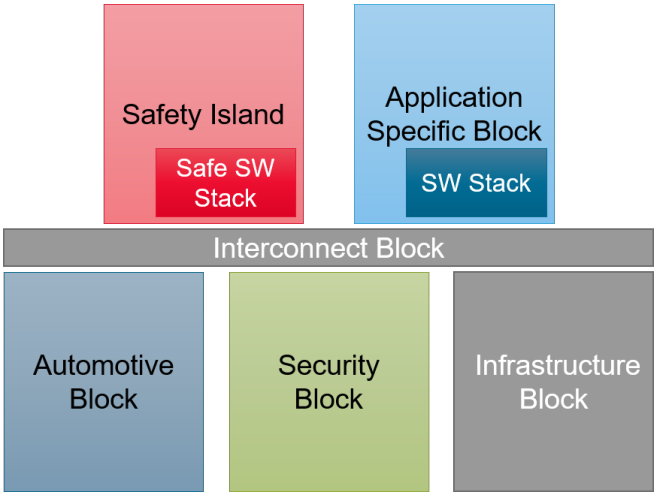

Figure 3.1: AutoSoC Functional Blocks

As it happens in most commercial solutions, the AutoSoC has two central processing units. The *Safety Island* is responsible for all safety-critical processing capabilities. It is composed of CPUs and memories that Safety Mechanisms must cover according to the requirements of ISO 26262. The division between safety-related hardware and the rest of the SoC components supports functional safety standards, as only the safety-related hardware must comply with ISO 26262. The other processing unit is the *Application Specific Block*. This unit implements the hardware required for application-specific processing. For example, it may include CPUs and memories for high-demand applications, GPUs, and image processing units for video applications. The target functionality for each given AutoSoC configuration will define the Hardware components required for the *Application Specific Block*. Also, it is essential to notice that the *Safety Island* and the *Application Specific Blocks* have dedicated software stacks; both can execute different Operational Systems and applications that will better suit their requirements.

The remaining blocks implement communication, security, and general SoC infrastructure. The *Automotive Block* is responsible for SoC communication with in-vehicle systems. The most common protocol deployed for in-vehicle communication is CAN. However, other options can be implemented, like FlexRay, LIN, Automotive Ethernet, among others. The *Security Block* is responsible for performing all security-related functionalities of the AutoSoC. The most common employment is cryptography cores, like AES and DES. However, we expect other security features to be explored. With this, the AutoSoC benchmark architecture allows future extensions to support the new security standard under development ISO 21434. The latter aims at defining a Cybersecurity Assurance Level (CAL), similar to the ASIL concept [46]. The *Infrastructure Block* is responsible for the online health monitoring of the SoC. It includes debugging features such as JTAG and UARTs to ease the development process. Finally, the *Interconnect Block* is

Table 3.2: Summary of Commercial SoC Analysis

	Amber	OpenRISC	Leon3
Processor	32-bit RISC	32/64-bit RISC	32-bit RISC
Instructions Set	ARM v2a	OpenRISC 1000	SPARC V8
Pipeline Configuration	Configurable 3 or 5 stages	Configurable 2 to 6 stages	7-stages
Memory	DDR3 interface (for Xilinx FPGA)	Single/Dual port RAMs	DDR/DDR2 ports
Main BUS	32-bit Wishbone	32-bit Wishbone	AMBA-2.0 AHB bus
Peripherals	Ethernet, UART, Timers, Interrupt Ctrl	fuseSoC library Opencores IP's: (e.g. CAN, AES, DES, ...)	GRLIB IP library
Virtual Platforms Support	ARM ISS	or1sim ISS (System C TLM Model)	Not Available
RTL Description	Verilog	Verilog	VHDL
Gate-level netlist	Optimized for FPGA	Synthesizable RTL	Synthesizable RTL
Software Stack	ARM gcc-toolchain	or1k-elf toolchain	Leon gcc-toolchain
Operational Systems	Not Available	Linux and RTEMS	Linux, RTEMs and VxWorks

responsible for internal SoC communication. It may deploy standard communication buses, like AXI and Wishbone, or more advanced options like a Network-on-Chip (NoC).

3.3. AUTOSOC BASE COMPONENTS

This section outlines the processing units, interconnect components, debug elements, and software workloads currently integrated into the AutoSoC. An initial configuration of the benchmark, named AutoSoC QM, is set up by deploying only the base components. The AutoSoC QM is a fully functional version of the benchmark and works as the foundation for further configurations. The modular design of the AutoSoC allows additional formats to be instantiated by simply enabling different Safety components. The following sections describe the available Safety components and AutoSoC configurations.

3.3.1. HARDWARE COMPONENTS

The selection of the CPU, as the central unit of the AutoSoC, considered different processor architectures, performance features (e.g., pipeline stages and memory interfaces), main buses, software stacks, and the possibility of development on multiple abstraction levels (Virtual Platforms, RTL, and gate-level). A further requirement is that the CPU has to be open-source. Table 3.2 summarizes the main attributes of the open-source platforms considered in this analysis.

Different analyzed options could be considered as promising candidates for the CPU. For instance, the Amber2 [47] is a 32-bit RISC CPU compatible with the ARM v2a instructions set. Another considered option was the Gaisler LEON3 [48]; it includes a seven stages pipeline, a comprehensive set of peripherals, and support scripts. This work has deployed the OpenRISC [49] (mor1kx implementation) as the main CPU. The OpenRISC includes a better variety of support tools, an active community, and the resources for developing a Virtual Platform. Also, the community supports a variety of compatible peripherals that can be easily integrated, including CAN, AES, and DES [50]. The OpenRISC community provides tools and examples for the development of SoCs. As part of that, there is an example SoC based on the mor1kx CPU. The package includes CPU, memory, UART, JTAG, and a debug unit; the components are connected with a Wishbone bus. Also, the example SoC contains a testbench with features for loading software applications to the memory and connection to the debug unit via JTAG. This example serves as a base for the AutoSoC. By deploying it, we can cover the infrastructure and interconnect blocks. Also, we can reuse part of the provided test environment to speed up the development.

3.3.2. SOFTWARE RESOURCES

One of the objectives of the Automotive functional safety analysis is to avoid disturbance of the safety-related functionalities of a system by *Random Hardware Faults*. In the case of an SoC, the software application executed by the CPU defines the functionality. For that reason, the software stack is an essential part of the functional safety analysis.

The current version of AutoSoC includes several software options. The intention was to integrate the available resources and the applications developed by ourselves in a unified repository in the AutoSoC simulation environment. The simulation of all available software applications is possible by suitably setting up the configuration files. AutoSoC includes several software resources organized by folders. For example, the Baremetal folder contains development resources as Makefiles, drivers, and around 50 compiled

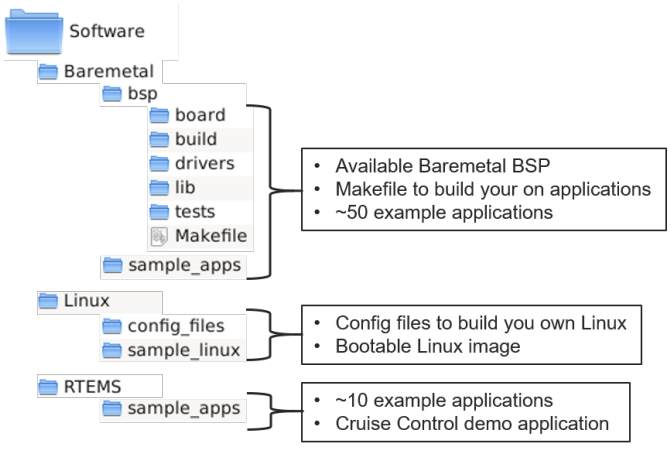

Figure 3.2: Software Resource Folder Structure

test applications. Also, the Linux folder contains a compiled Linux kernel (bootable in simulation) and the configuration files for building customized Linux configurations. Furthermore, the RTEMS folder includes a development environment, including Makefiles and drivers, enabling buildings of new RTEM applications. Figure 3.2 illustrates the software folder structure available in the AutoSoC package. It is essential to highlight that all software resources are compatible with the simulation environment, also included within the benchmark.

Finally, the AutoSoC also incorporates an Automotive Cruise Control application. The development of such an application aims to increase the benchmark representation; by including an actual automotive application, we can leverage the functional safety analysis considering fundamental functionalities. The Cruise Control application deploys the RTEMS Operational System and comprises four real-time tasks: the Sensor reads vehicle sensor data through the AutoSoC CAN bus; the Control computes actuation parameters based on driving configurations and the data retrieved by the Sensor; The Engine set the engine parameters via an Analog-Digital Converter (ADC); and, the House Keeping monitors the correct functioning of the software application. Figure 3.3 illustrates the control flow of the Cruise Control application.

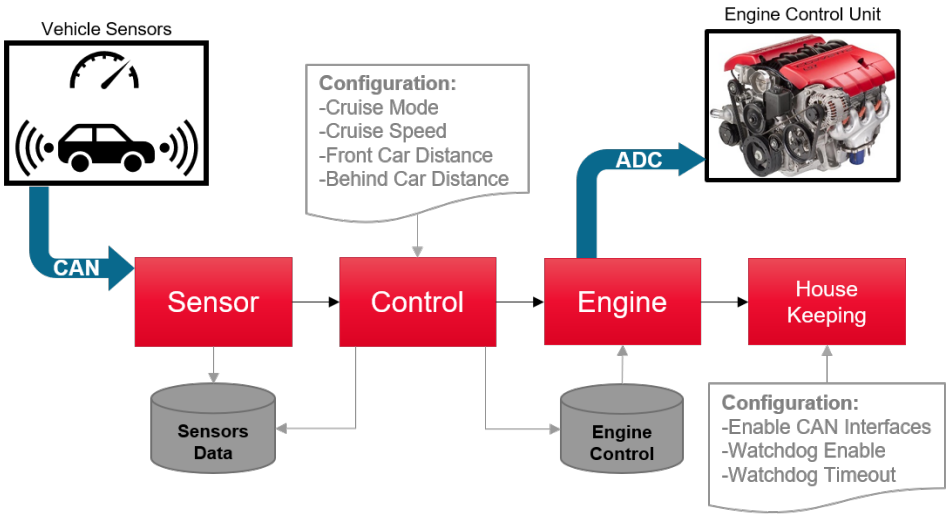

Figure 3.3: Cruise Control Application

3.3.3. SIMULATION ENVIRONMENT

The elements previously described are integrated by a simulation environment; such an environment deploys Makefiles to automate the configuration and simulation of the AutoSoC. The execution of a complete fault injection campaign is possible by the execution of three commands:

1. **make elab**: controls the elaboration of the AutoSoC version; the setup is based on configuration files that enable features of the benchmark.

2. **make good_sim**: starts the fault-free simulation of the design previously elaborated; the simulation will dispatch the software application selected in the campaign configuration files.

3. **make fault_sim**: begins the fault injection campaign; all FI simulation parameters are configurable via configuration files (e.g., fault type, fault injection time, strobes, available resources for parallelization).

At the end of the FI campaign, the Makefile retrieves the result of each simulation and presents a summary of the fault space classification. Figure 3.4 illustrates the simulation environment integrated in the AutoSoC benchmark package.

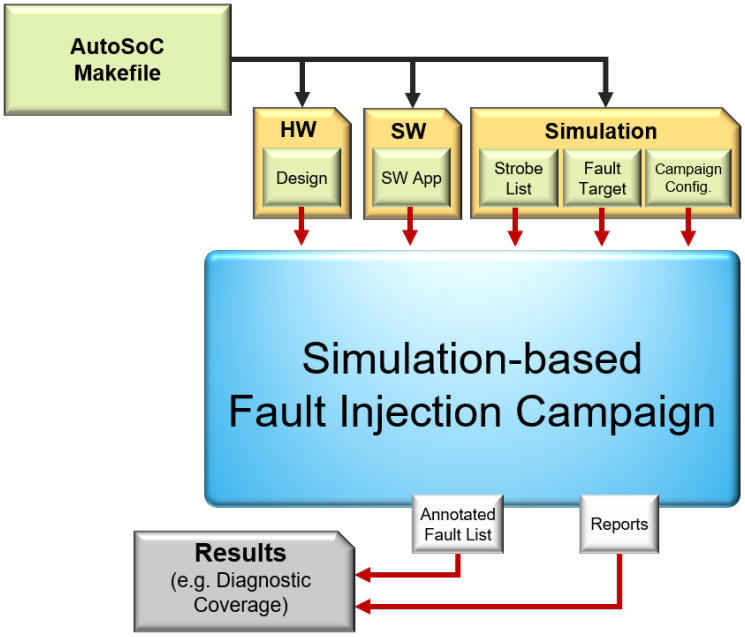

Figure 3.4: AutoSoC Simulation Environment

3.4. AUTOSOC SAFETY COMPONENTS

Another critical aspect of the benchmark is the availability of Safety Mechanisms in the Safety Island. This block is responsible for executing safety-critical applications; therefore, we need to ensure that potential faults can be detected, avoiding possible harm to the expected functionalities. The CPU, as the primary unit of the Safety Island, is the primary target for the safety evaluation. Hence, different safety mechanisms schemes were conceived, targeting different Automotive Safety Integrity Levels (ASIL).

3.4.1. DUAL-CORE LOCKSTEP

The first option deploys time diversity Dual-Core Lockstep (DCLS) as the main Safety Mechanism. The DCLS configuration includes a redundant copy of the CPU, delay units for time diversity, and compare units for fault detection. Figure 3.5 illustrates the implementation of the DCLS with time diversity.

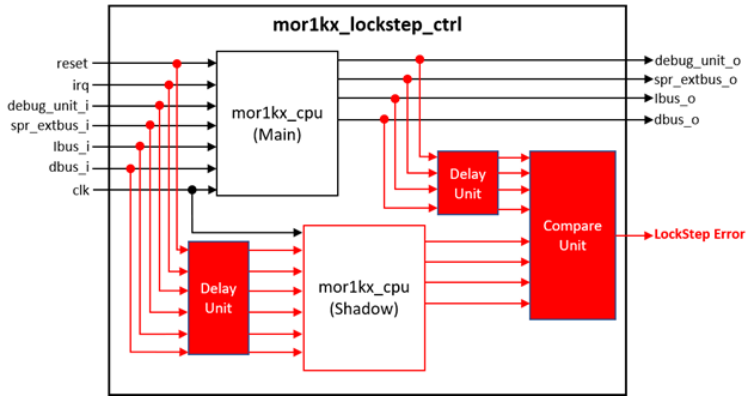

Figure 3.5: Time diversity Dual-Core Lockstep implementation

The performance of the central processor is not affected by the DCLS implementation. The main CPU is the only one with write access to the bus, controlling the functionality of the SoC. On the other hand, the shadow CPU does not perform any write access to the SoC resources. Instead, the outputs of the shadow CPU are used only by the Compare Unit for fault detection. In case of a mismatch between the results of both processors, an alarm is activated by the Compare Unit. Despite the additional fault coverage by including DCLS, we still need to consider the effect of common-mode failures that can impact both processors and are not detectable by comparison of their outputs [51]. The DCLS mechanism also includes time diversity, minimizing the potential of common-mode failures. Time diversity works by applying a delay in the execution of the shadow processor. The time difference is obtained by including a delay unit in the driven signals of the CPU. Delay units are also added to the outputs of the central processor to align both core outputs for the Compare Unit. The time shift applied by the Delay Units can be configured; the current version applies a delay of 2 clock cycles to all signals. The shadow CPU execution delay configuration must consider the system requirements for maximum fault tolerance time. Since this delay is also applied to the inputs, a mismatch will be detected only after the configured delay.

Dual-Core Lockstep is the most used SM scheme for processors targeting ASIL D applications. However, not all applications demand ASIL D and the extra cost of including a redundant copy of the CPU. For that reason, AutoSoC incorporates additional configurations targeting different ASIL requirements.

3.4.2. SOFTWARE TEST LIBRARIES

A Software Test Library (STL) is a collection of software tests targeting the detection of permanent faults; these can occur anytime during the execution of a safety application

and can lead to safety goal violations. An STL corresponds to a set of software procedures, usually developed in assembly code, C code, or both. These may be executed either at boot-time or run-time. In the former case, they require supervisor capabilities to avoid conflict with the Operating System (OS). On the other hand, the deployment of STLs at run-time requires symmetry with the OS. For that reason, it is essential to make these tests run in short periods; usually, a few milliseconds, avoiding affecting the behavior of the other software applications running on the same hardware. Therefore, the software scheduler will dispatch these tests at specified time intervals when the hardware is idle or running less time-sensitive applications.

Several semiconductor and IP companies started to provide their customers with STL solutions for online fault detection in recent years. The advantage stemming from their adoption lies first of all in the fact that system companies can test their products in the field while guaranteeing a given fault coverage, even without knowing the implementation details (black-box testing). Moreover, STLs perform the test precisely in the system operating conditions, thus executing at speed and avoiding any overtesting. Finally, they do not require any change in the hardware, thus avoiding any area or performance overhead. On the other side, the generation of STLs is mainly manual and requires special skills to achieve sufficiently high fault coverage figures. Several recent works introduced guidelines on how to correctly generate STLs for CPUs [52], [53] and peripherals [54], how to speed up their tests [55], how to maximize their fault coverage in the different scenarios (possibly minimizing the test time [56]), and how to re-use existing STLs.

3.4.3. INTERNAL MEMORIES ECC

Usually, in complex CPUs, internal memories occupy the highest area on the physical device. As the component size is directly related to the probability of faults, internal memories are a primary target for SMs. The ISO 26262 standard includes recommendations for well-known memory Safety Mechanisms. Based on the guidance and the findings of the industry solutions characterization, Error-Detection-Correction Code (ECC) was selected as an option to protect the internal memories of the CPU. The current implementation of the Safety Island CPU includes seven blocks of internal RAMs. Together, the internal memories represent 91,3% of the total fault targets in the RTL representation of the CPU. Therefore, deploying an SM with high Diagnostic Coverage, like ECC, on all internal memories will provide a satisfying coverage for the overall CPU.

3.4.4. EXTERNAL MEMORY ECC

Other elements that comprise the Safety Island are also a target for *Random Hardware Failures*; therefore, these elements must undergo functional safety verification for the possibility of single points of failure. Generally, software applications must be loaded to the external memory to be executed by the CPU. Also, the applications utilize memory for storing data and controlling parameters. As the software application function relays on the external RAM, memory failures directly impact the intended functionality. ECC must also cover the external RAM to avoid propagation of internal memory faults to the outputs of the Safety Island.

3.4.5. BUS PARITY

The data bus is responsible for data transmission between the memory and the CPU. For that reason, a fault in the data bus could propagate to the CPU or the memory and would not the detected. The addition of a parity checker verification to the bus can avoid these cases and protect data transmissions between CPU and memories. The parity checker monitors data bus transmissions and calculates a parity bit for all communications between CPU and memory. The control bit utilizes a dedicated connection for communication between the parity check blocks. In case of a wrong parity, an alarm is set to inform the system.

3.4.6. CHECKPOINT CONTROL

Even if the DCLS is employed, both CPUs could get stuck in the same software instruction, and none of the mentioned SMs could detect this fault. For that reason, we include an SM to verify the correct execution of the software applications; named Checkpoint Control. The Checkpoint control monitors the Data Bus expecting pre-determined software signatures in specific memory locations. Thus, the mechanism works as a hardware watchdog, but instead of expecting a single refresh from the software application, it expects a different signature for each software task. Consequently, the SM can verify if the software application is running and if the control flow is as expected. Furthermore, the Checkpoint Control is fully customizable during elaboration, allowing the definition of the software signatures, expected signatures, and deadlines.

3.4.7. SAFETY MONITOR

Finally, we developed a Safety Monitor to integrate all the detection alarms. In the case of fault detection of any SM, the Safety Monitor generates an external alarm and an error code to indicate where the fault was detected. Figure 3.6 illustrates the architecture of the AutoSoC Safe configuration, including the DCLS, External Memory ECC, Bus Parity, and Checkpoint Control.

3.5. AUTOSOC CONFIGURATIONS

This section outlines the available benchmark forms and how they can be set up by enabling the different safety components. The available configurations comply with the Functional Blocks: Safe, Automotive, Infrastructure, and Interconnect. The Application Specific and Security Blocks, as illustrated in Figure 3.1 will be developed in the following stages of our work. The modular design of the AutoSoC allows the reuse of the functional safety analysis on later configurations.

As part of its modular concept, several configurations of the AutoSoC are possible by enabling different combinations of the mentioned components. For defining a new configuration, based on the provided simulation folder, the user must select the hardware components in the elaboration config file, enabling any combination of Safety Mechanisms by adding defines to the *'plus args'* config file (e.g., +define+DCLS). The new configuration can then be elaborated and simulated with the provided Makefile. The benchmark enables the creation of multiple formats by combining the available component; nevertheless, we have defined a group of initial configurations for the AutoSoC. These

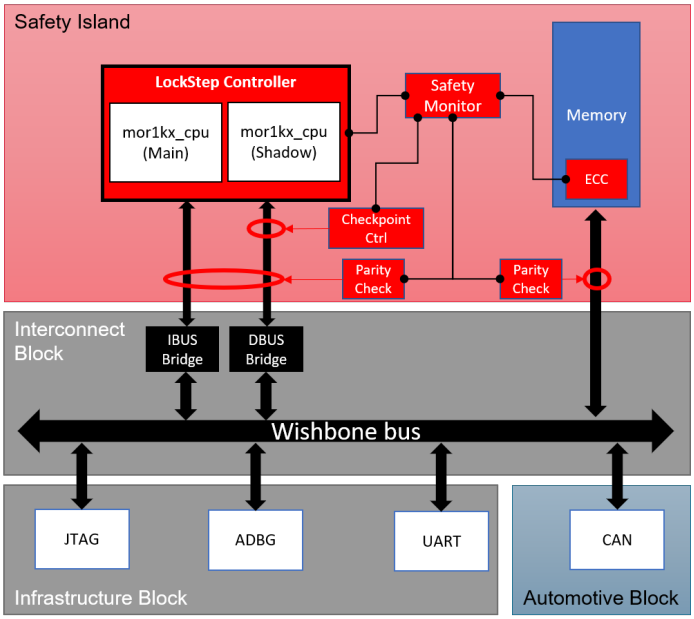

Figure 3.6: AutoSoC Safe Configuration

configurations follow industry standards when considering common SM combinations. Table 3.3 illustrates some potential configurations for the AutoSoC. For the scope of this work, we have performed a preliminary safety assessment for three configurations; the analysis targets AutoSoC ECC, AutoSoC STL, and AutoSoC DCLS as candidates to different ASIL levels.

Table 3.3: AutoSoC Configurations

Benchmark Configurations	Dual Core LockStep	Internal Mem ECC	Software Test Libraries	BUS Parity	Checkpoint Control	Safety Monitor
AutoSoC QM	-	-	-	-	-	-
AutoSoC ECC	-	+	-	-	-	-
AutoSoC STL	-	+	+	-	-	-
AutoSoC DCLS	+	-	-	-	-	+
AutoSoC SAFE	+	-	-	+	+	+

3.6. PRELIMINARY FUNCTIONAL SAFETY ANALYSIS

This section describes the functional safety analysis of the AutoSoC configurations. As specified by ISO 26262, functional safety analysis aims to decrease the risk of failures caused by malfunctions. Within electronic systems, it focuses on avoiding that *Random Hardware Faults* can disrupt the expected functionality of a design. The ASIL defines the required risk reduction for a particular functionality. Functionalities with a higher risk of

hazard situations demand a higher ASIL. In general, to reduce the risk of malfunctions induced by random faults, we include SMs. The required percentage of detection, or DC, is defined by the ASIL.

Typically, functional safety analysis occurs at later stages of the hardware design. Hence, additional parameters like area, Failure-in-Time (FIT) rate, and Failure Modes distribution are necessary to confirm design compliance to the required ASIL. Furthermore, we need these parameters to calculate Safety Metrics that show the design capacity to cope with different fault models. For that reason, the current AutoSoC analysis is considered preliminary. The following chapters determine the possible failure modes, define the diagnostic coverage based on the failure mode distribution, and calculate the final safety metrics.

3.6.1. AUTOSOC DCLS CONFIGURATION

Hardware redundancy schemes, like dual-core lockstep, are defined by ISO 26262 as recommended safety mechanisms for processing units. The standard defines the typical diagnostic coverage for these mechanisms as high, meaning 99% of detection for random hardware faults. Therefore, the implementation of DCLS should aim to provide early detection of failures by step-by-step comparison of results produced by two processing units operating in lockstep. The AutoSoC DCLS configuration intends to comply with the description from ISO 26262. Also, the implementation of time diversity increases the DCLS features by addressing the effects of common-mode failures.

A preliminary investigation of the mor1kx_cpu description shows a potential of 337.752 possible fault targets. If we consider the SA0 and SA1 fault models, as required for ISO 26262 permanent faults analysis, there are a total of 675.504 faults to be analyzed. The DCLS safety mechanism intends to identify faults in the mor1kx_cpu. By respecting the Diagnostic Coverage defined by ISO 26262 for the DCLS, we can assume that the Lockstep Controller will detect 99% of the faults in the mor1kx_cpu(Main). With 99% of fault coverage, we can expect the AutoSoC DCLS to be a good candidate to comply with ASIL D requirements. Table 3.4 illustrates the potential fault coverage for the AutoSoC DCLS configuration.

Table 3.4: DCLS CPU Fault Coverage

Fault Target	SA(1/0) Faults	Detected by DCLS	Residual Faults
mor1kx_cpu	675.504	668.749	6.755

3.6.2. AUTOSOC ECC CONFIGURATION

As described for the Processing Units, ISO 26262 also includes recommendations of SMs for volatile and non-volatile memories. One of the recommendations is the deployment of memory monitoring using Error-Detection-Correction Codes (ECC). Traditionally, ECC algorithms can detect one and two-bit failures and some three or more bit failures in a word. The standard defines the typical diagnostic coverage for ECC as 99% of detection for random hardware faults. Usually, on complex CPUs, internal memories or caches occupy the largest physical devices area. For that reason, a high detection rate on

the memories will provide a significant contribution to the design Failure-In-Time (FIT) rate. This contribution will appear in the Failure Modes (FM) distribution, with cache-related FMs requiring SM to decrease the residual FIT. In addition, it is a common design practice to protect the cache memories with ECC or Parity.

In the AutoSoC design, the internal memories represent 633.344 possible fault targets considering the SA0 and SA1 fault models. This number represents 93,7% of the total number of fault targets for the entire CPU. For that reason, the addition of SM to the internal memories means an excellent overall coverage for the CPU faults. The AutoSoC internal ECC configuration considers the incorporation of ECCs to all internal memories. Table 3.5 demonstrates the fault coverage of the ECC for each inner memory block. The total number of faults covered by the ECCs, considering the 99% DC defined by ISO 26262, is 627.011 faults. This coverage represents a 92,8% Diagnostic Coverage of the entire CPU. These figures acknowledge the AutoSoC internal ECC configuration as an excellent candidate to comply with ASIL B requirements.

Table 3.5: Internal Memories ECC Fault Coverage

Fault Target	SA(1/0) Faults	Detected by ECC	Residual Faults
Fetch instructions cache ram	262.144	259.523	2.621
Fetch instructions cache tag ram	20.992	20.782	210
Fetch instructions MMU ram	8.192	8.110	82
Load/Store data cache ram	262.144	259.523	2.621
Load/Store data cache tag ram	19.968	19.768	200
Load/Store data MMU ram	8.192	8.110	82
Load/Store store buffer	51.712	51.195	517
TOTAL	**633.344**	**627.011**	**6.333**

3.6.3. AUTOSOC STL CONFIGURATION

In several cases, the overhead required by redundancy schemes as DCLS is not possible. For that reason, there is an increasing demand for alternative strategies for the online testing of automotive processors. This section describes the main characteristics of the software test libraries under development to improve the AutoSoC CPU fault coverage and reports the preliminary results.

We gather preliminary results on two AutoSoC CPU modules: the Arithmetic Logic Unit (ALU) and the Load and Store Unit (LSU). The STL programs have been developed resorting to three of the most common strategies for Software-Based Self-Test (SBST) generation [57]: ATPG-based, deterministic, and evolutionary-based [58]. The current STL comprises 16 test programs for a total of 64 KB. The AutoSoC STL Configuration targets the CPU (mor1kx_cpu), cleared of all the possible sources of non-determinism such as Instruction Cache and Data Cache. Furthermore, it is essential to note that a controlled STL execution must consider a deterministic stream of instructions entering the pipeline. For that reason, modules that lead to a fluctuating in the control flow (e.g., caches) should be deactivated for the fault grading process. This scenario does not prevent caches (or similar) components from being used when the STLs are deployed; however, it requires additional preparation to execute the test libraries.

For evaluation of the STLs, we carried out FI simulation on 42.160 faults targeting

the mor1kx_cpu at RTL and a total of 60.672 permanent faults for the mor1kx_alu and the mor1kx_lsu units at gate-level. When considering the mor1kx_alu and mor1kx_lsu, there are 4.938 fault targets. The FI experiments were performed at both the RTL and gate-level, mimicking the typical industry flow; RTL estimations work as a proxy for the gate-level fault coverage estimation, guiding the STL development process. Additionally, the analysis of Safe fault by formal methods reveals a non-negligible increase in the fault coverage of the two targeted modules. Table 3.6 sums up the gathered results showing the achieved fault coverage on the ALU and LSU modules, both at the RTL and gate-level. The table also differentiates the total Fault Coverage (FC) from the Testable Fault Coverage (TFC), as TFC considers the fault coverage after excluding Safe faults.

The deployment of software routines to identify permanent faults is shown to be effective in multiple units of a CPU [59]. Although it is not always possible to achieve ASIL D fault coverage requirements by deploying STLs, they are an appealing alternative when combined with other SMs. A common practice in the automotive industry is to correlate STLs with ECC in the internal memories of the CPU. For instance, in [59] the authors achieved a permanent fault coverage of 84,4% by deploying an STL in an OpenRISC CPU similar to AutoSoC CPU. The AutoSoC CPU contains 42.160 targets for SA0 and SA1 faults, not considering the internal memories. If we believe the fault coverage from [59], the STL would be able to detect 35.583 faults; including the STL routines in the AutoSoC ECC Configuration 3.6.2, the SMs combination would detect 662.594 faults. As the total number of faults is 675.504, the combined detection rate represents a Diagnostic Coverage of 98%. This figure would allow the combination of the AutoSoC STL and ECC configurations to be a good candidate to comply with ASIL C requirements.

Table 3.6: Selected CPU Modules STL Fault Coverage

CPU Modules	RT-Level		Gate Level	
	FC [%]	TFC [%]	FC [%]	TFC [%]
ALU + LSU	68,71	80,04	76,23	85,43

3.7. CONCLUSIONS

The development of autonomous vehicles is driving the industry to close the technological gap demanded by these applications. The research community is proposing solutions to address safety, security, performance, among others. However, it may be hard to assess the quality of their results. In most cases, there is limited access to representative designs, and comparison with industrial methodologies is very complicated. To address this matter, we present the AutoSoC benchmark suite. Our work intends to provide researchers with an SoC based on commercial solutions, includes all essential components, is highly customizable, and allows comparability between distinct methodologies and results.

This chapter outlines the current architecture options incorporated in the AutoSoC, including hardware components, software applications, operating systems, and safety mechanisms. Also, we describe a preliminary functional safety assessment targeting different ASIL configurations. Finally, the following chapters deploy the AutoSoC as a test case for evaluating the proposed methodologies; named, targeting early estimation of

safety metrics, improvements to online fault detection, and enhancements to the fault space analysis. Additionally, we believe that the availability of this benchmark suite will allow researchers to develop new solutions and quantitatively assess their effectiveness, thus contributing to the advancement of the state-of-the-art in the related areas.

4

EARLY ESTIMATION OF DESIGN SAFETY METRICS

The requirements of ISO26262 for developing safety-critical Integrated Circuits (IC) demand substantial efforts on fault analysis for safety metrics evaluation. Failing to achieve the required conditions entails modifications to the circuit, additional iterations through critical design phases, and consequently extra costs and delays. For that reason, providing accurate methods to estimate safety metrics is of great importance. In this chapter, we introduce a methodology that can efficiently and precisely evaluate the safety metrics of Automotive designs. The technique is based on the characterization of the hardware description in different abstraction levels to determine how the components contribute to fault propagation. Also, by examining the test stimuli applied during simulation, we can rank Workloads/Testbenches according to their fault detection coverage. We validate the approach by running fault injection campaigns on distinct hardware designs in RTL and gate-level. Our results show that the fault detection coverage can be estimated with an average error rate of 3% at up to 20X faster execution times when compared to the traditional campaigns. Hence the methodology provides an efficient and cost-effective mechanism to support engineers in a confident design space exploration.

Parts of this chapter have been published in the IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), 2021 [60].

FUNCTIONAL Safety Verification, as defined by ISO26262, is usually performed at later stages of the development cycle; when failing to achieve the required safety metrics demands additional iterations through critical development and verification phases. Such a scenario has a high impact on costs and development time. In a typical lifecycle, the safety concept and architecture are established at the early development stages, requiring engineers to estimate fault detection coverage without a proper evaluation methodology. As a result, a misleading architecture decision before implementation will be exposed only at the final stages of the development when modifications are expensive. For that reason, there is a high demand for techniques that can support safety engineers with design space exploration of safety features, increasing the confidence in conceptual decisions and avoiding rework.

The recent advances in fault classification propose optimizations in fault analysis and accentuate the strengths of several different technologies. However, the deployment of such techniques assumes that the design is at the final development stages; for that reason, they are not fully applicable for early estimation of the safety metrics. Furthermore, early design exploration of safety metrics is a gap in the development lifecycle. Design space exploration refers to the activity of exploring design alternatives before implementation. The concept is established in several domains of IC development, like area, performance, power consumption [61][62], high-level synthesis [63], deep learning [64], among others. Nevertheless, to the best of our knowledge, there are no methodologies for early design space exploration targeting safety metrics for compliance with ISO26262.

Our work introduces a methodology for estimating safety metrics in Automotive designs. By allowing engineers to evaluate fault detection rates before the final development stages, we provide a tool for the design space exploration of safety architectures, improving the confidence in conceptual decisions and decreasing the chances of rework. The methodology is based on the hardware components' characterization and the analysis of the test stimuli concerning fault propagation; the technique can be deployed using RTL or gate-level, covering multiple steps of the development lifecycle. Initially, we identify the prime propagation nodes of the hardware design; these are sequential elements that can hold the effect of faults. Then, we designate a weight for each prime node that represents the number of components where a fault can propagate to such node; in other words, the elements in the Cone of Influence (CoI). Next, we need to understand the behavior of the prime propagation nodes under the influence of faults; for such, we perform FI simulation only on these nodes. Finally, we can assess fault space analysis results by compiling the prime propagation nodes' weight and their faulty behavior. Additionally, we consider the impact of fault activation by analyzing constant nodes during the fault-free simulation; such a technique allows the classification of the Workload/Testbench by their fault detection rate potential, resulting in enhanced estimations for the analysis of gate-level designs. The main contributions of this work are:

- A systematic approach for the estimation of safety metrics of Automotive designs in multiple phases of the development lifecycle;
- An effective method to rank Workloads/Testbenches according to their impact in the fault detection coverage;

- Validation of the methodology in Automotive IPs, considering RTL and gate-level abstraction levels;
- The results estimate the fault detection coverage with an average error of 3% at up to 20X faster execution times.

4.1. SAFETY METRICS ESTIMATION METHODOLOGY

Fault analysis targeting ISO26262 compliance aims to identify faults that can propagate to safe-related outputs of the system. As these faults may disturb a safety goal, the design should include mechanisms to detect and control them, maintaining a safe state. The propagation of the fault effect depends on the hardware characteristics and the stimuli applied to the design. These two aspects are fundamental to understanding a design's behavior under the influence of faults. Additionally, as previously discussed, the classification of the Fault Space is mandatory for calculating the safety metrics; for that reason, forecasting fault behavior is crucial for early estimation of the safety metrics. One of the essential parameters for calculating the Diagnostic Coverage (DC) is the number of Fault Space elements classified as Detected. The DC represents the capability of the design to cope with *Random Hardware Faults*; for that reason, such a value is critical for calculating the SPFM, LFM, and PMHF, as described in 1.2.3.

Aiming to predict the number of Detected faults in different phases of the development lifecycle, and without the need to execute an entire Fault Injection campaign, we propose the Safety Metrics Estimation methodology. Initially, the structure of the hardware design, in the available abstraction level, is analyzed to identify components that can retain the effect of faults; these are, in general, sequential elements, inputs, and output ports; these elements are named prime propagation nodes. Next, a weight is calculated for each prime propagation node; the weight represents the number of hardware elements where a fault can propagate to such a node. After, we deploy simulation to identify the effect of the workload regarding fault activation and fault propagation; such a step is crucial to improve the prediction of fault classification. Finally, by combining design characterization with the contribution of the workload, we can estimate the number of Detected faults for a given hardware design and workload.

The Safety Metrics Estimation methodology deploys four distinct phases: Design Characterization, Fault Propagation Analysis, Fault Activation Analysis, and Estimation of Fault Injection Results. Figure 4.1 illustrates each phase. The next sections describe the technologies and activities deployed in each phase.

4.1.1. DESIGN CHARACTERIZATION

The design characterization aims to understand how each fault in the Fault Space propagates in the design; it evaluates fault propagation to selected sequential elements, creating a matrix representing the potential of faults affecting each component. Such a matrix allows the calculation of a weight denoting all fault nodes as a function of the sequential elements. The design characterization is based only on the physical attributes of the hardware; therefore, the influence of the test stimuli, or workload, is not contemplated at this stage. For that reason, formal methods appear as a good candidate for the technology to extract the required information from the design. This work deploys the

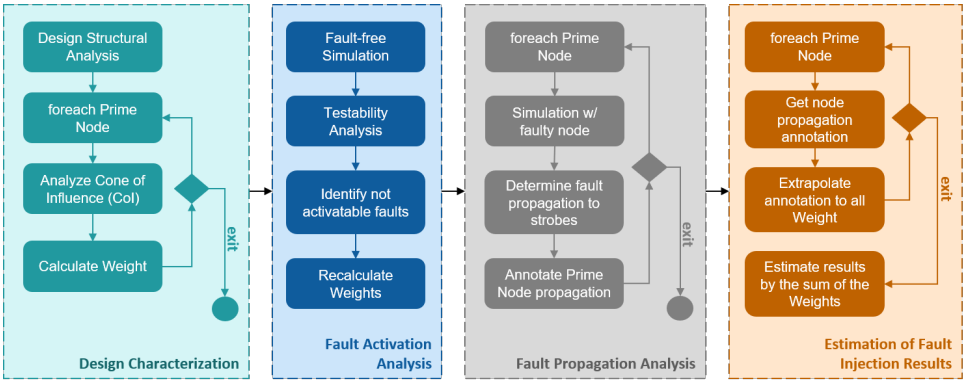

Figure 4.1: Safety Metrics Estimation Flow

Functional Safety Verification (FSV) application from Cadence®JasperGold (JG) Formal Verification Platform.

ISO26262 requires that the Fault Space analysis contemplate all cell ports on the gate-level representation of the design. However, our methodology intends to support early estimation of safety metrics, so we need to adjust this requirement for higher hardware abstraction levels, such as RTL. State-of-the-art EDA tools enable the evaluation of the Fault Space at RTL; for such, they can identify constructions of description languages like Verilog by the intended functionality and determine if they will become combinational or sequential logic in the gate-level representation. Even though faults in RTL and gate-level may not directly correlate, as synthesis will influence the resulting gate-level, the implemented functionalities must be the same; for that reason, the RTL prediction is a good indication of the faulty design behavior.

As illustrated in Figure 4.1, the first step of the design characterization is the Structural Analysis; the analysis is described in 1.3.1. Jasper Gold FSV enables the analysis in RTL and gate-level; it also identifies RTL constructions as sequential or combinational components. Therefore, as a result of the Structural Analysis, we can identify every element of the design and extract information about the Fault Space. First, the Structural Analysis initiates with the fault target configuration; for this work, we have specified all cell ports for SA0 and SA1 fault models. Next, the fault list is optimized by the identification of Safe faults. As previously discussed, formal methods can prove that Safe faults cannot propagate to design outputs; therefore, these faults can be ignored, improving the fault propagation estimation.

The next step is the analysis of the Cone of Influence (CoI) of each prime propagation node. The prime propagation nodes are the hardware components identified as sequential elements and the output ports of the design; the selection of these nodes considers their possibility to retain the effect of faults and the potential for correlation between RTL and gate-level [65][66][67]. Furthermore, the CoI details all the fault nodes physically connected to a given hardware element. The CoI analysis is generally deployed on the design outputs to identify faults that cannot propagate to them; however, this work deploys the CoI analysis on every prime propagation node to understand the prop-

agation of faults to such hardware components. For each element classified as a prime propagation node, the flow continues as follows:

1. Extract the fault nodes inside the CoI;

2. Remove sequential elements as they are part of the prime propagation nodes list and will be computed in separate;

3. Compute all remaining fault nodes.

After analyzing all prime propagation nodes, we have collected the required information to calculate the weights. The weight represents the number of faults propagating to a given prime propagation node. For example, a fault node that propagates to only one flip flop has a weight contribution of one. Faults nodes with a physical propagation path to multiple flip flops, need to have their weight contribution calculated based on the number of flip flops they can affect. Figure 4.2 illustrates the weight calculation on an example circuit.

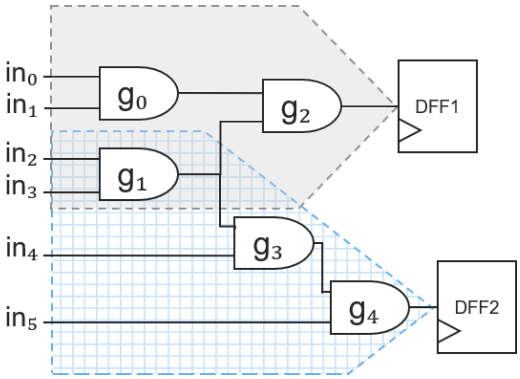

Figure 4.2: Flip Flop Cone of Influence example

Figure 4.2 shows an example design containing Flip Flops (DFF) and gates (g). The figure also highlights each flip flop's Cone of Influence (CoI). The solid pattern represents the CoI of 'DFF1', while the checkered pattern represents the CoI of 'DFF2'. The gate 'g1' is inside both CoI, implying that a fault in its ports can affect 'DFF1' and 'DFF2'. As previously described, a fault node that propagates to only one flip flop has a weight contribution of one. Considering that each gate (g) contains three fault nodes, we can calculate that 'DFF1' has a starting weight of six, from the faults in 'g0' and 'g2', while 'DFF2' has a starting weight of 6, from the faults in 'g3' and 'g4'. To calculate the weight contribution of 'g1', as it is inside multiple CoI, we need to divide the number of fault nodes by the number of flip flops they can affect. In the example, as 'g1' is inside two CoI, any fault node in the gate will have a weight contribution of 0,5. Consequently, 'g1' will induce additional 1,5 to 'DFF1' and 'DFF2' weights. It is essential to highlight that the described analysis considers a single fault model; therefore, we must repeat it for every fault model in the Fault Space. Finally, considering a single fault model, the

netlist characterization of the example design would result in a final weight of 7,5 for both 'DFF1' and 'DFF2'.

The design characterization will result in a representation of the fault propagation potential of a given circuit, by only their prime propagation nodes (flip flops and primary outputs). Each prime propagation node will have a weight that expresses the number of faults factored by the node. Therefore, we can estimate the classification of all fault targets in the design by analyzing fault effects in the prime propagation nodes. After understanding how the physical characteristics of the circuit impact in the fault behavior, it is necessary to compute the contribution of the workload regarding fault activation and propagation.

4.1.2. FAULT ACTIVATION ANALYSIS

The workload has an essential role in determining the behavior of a design under the influence of faults. The stimuli applied by the Testbench for simulation of the circuit determine if a fault node will be activated. The combination of test inputs will determine the logic value on the gate ports; if such a value is constant, a stuck-at-fault with the same logic value will never be activated. For example, if a given hardware component has a continuous logic value of '0' due to the applied test stimuli, the effect of an SA0 fault will never be noticed. As the fault would never propagate in this example, we could remove it from the previously calculated weight, increasing the accuracy of the prediction.

To verify the activation of faults, we need to execute a behavior simulation of the design to a given workload; by monitoring the logic value from all hardware components throughout the simulation, we can identify constant nodes, and therefore, not activatable faults. The chosen technology for this analysis step is FI Simulation. Even though this step does not require fault injection, we can deploy the fault-free simulation to extract the constant analysis. Our work deploys Cadence®Xcelium™Fault Simulator (XFS) to represent this technology.

The test stimuli applied to the circuit will determine if a fault will be activated. Hence, to identify fault nodes that are not exercised by the current workload, we need to analyze the circuit's inputs during the simulation. The fault activation analysis starts by identifying test inputs that are constant during the design simulation. Then, this information is applied to perform the testability analysis. The testability analysis identifies faults that are unobservable for a given workload. Figure 4.3, illustrates an example of fault activation analysis.

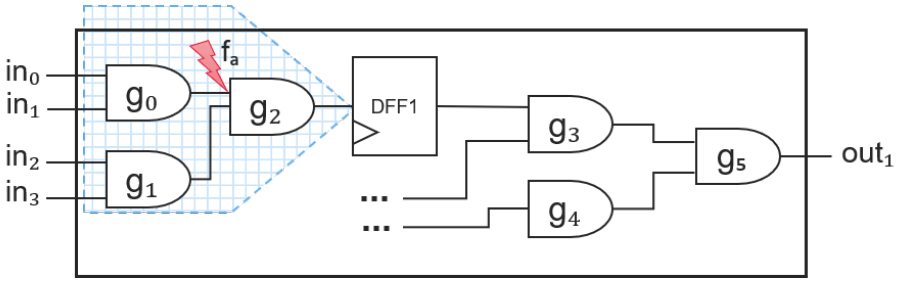

Figure 4.3: Fault Activation example

In the example illustrated in Figure 4.3, a fault node 'fa' depends on the logic values applied to 'in0' and 'in1' to be activated. Any combination of the inputs that produce a logic value '0' in the gate 'g0' output, results in an unobservable SA0 fault in 'fa.' The same would be true for an SA1 fault if 'g0' outputs a constant logic value '1'.

By examining all test inputs and identifying constant signals throughout the circuit, it is possible to classify several fault nodes that are unobservable for a specific fault model. As the workload never activates these fault nodes, we can conclude that they will never propagate to circuit outputs during FI Simulation. Therefore we can determine that these faults will be Undetected during the Fault Space analysis with the current workload.

After identifying all unobservable fault nodes for a given workload, we need to consider them for the weight calculation. All faults identified as unobservable will have a weight contribution of zero. Consequently, we can recalculate the weight from the design characterization step with the inputs from the activation analysis. The resulting weight enhances the results from the design characterization by assessing the role of the workload regarding fault activation.

4.1.3. FAULT PROPAGATION ANALYSIS

Likewise, the stimuli applied by the workload determine if a fault injected in a given node can propagate to the outputs of the design; the test stimuli may put hardware gates in a logic state that can mask the propagation of faults. For that reason, fault propagation analysis is crucial to determine the behavior of a faulty design. Moreover, as previously discussed, the Fault Space analysis demands FI simulation on every cell port of a hardware design. Nevertheless, in the previous phases, we have defined a method to create a representation of the fault space of a given circuit by only its prime propagation nodes; therefore, we can estimate the Fault Space analysis results by only simulating faults in such nodes. Furthermore, by decreasing the number of simulations, we can significantly improve execution times and enable early estimation of the safety metrics.

In the previous phases, we have determined a weight for each prime propagation node; it represents the number of hardware components where a fault could reach such a node. Next, we need to assess the behavior of a fault injected in the prime propagation nodes; the annotation of such a fault can be extrapolated based on the weight, predicting the behavior of the other faults without additional simulations. Initially, we need to define the fault target for each prime propagation node; in the case of gate-level designs, the target will be the output port of flip flops; in RTL, the target will be the sequential variable itself. Next, we deploy FI Simulation to determine the propagation of each prime propagation node. The FI campaign follows the exact arrangement as described in 1.3.1; the only difference is that the Fault Space is manually adapted to consider only the prime propagation nodes. Figure 4.4 shows an example of the fault propagation analysis.

The example illustrates the propagation of a fault 'fo' injected in the output port of 'DFF1' to the circuit output 'out1'. The observability of the effects of 'fo' in 'out1' depends on the logic level of the gates 'g3' and 'g5'. As the workload is responsible for setting the logic level of these gates, the simulation can confirm the fault propagation to the output. Finally, at the end of the FI campaign, we can comprise each prime propagation node's weights and fault annotation to estimate the classification of all faults in the hardware

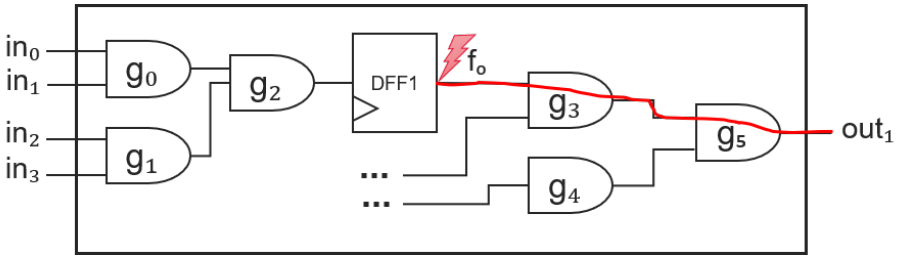

Figure 4.4: Fault Propagation example

design.

4.1.4. ESTIMATION OF FAULT INJECTION RESULTS

The final phase is the estimation of fault injection results. By considering the classification of the fault effects on the prime propagation nodes and their weight, we can estimate the fault classification to the entire circuit. For such, we extrapolate the FI result to all faults represented by the weight; therefore, if a node weighted five and was classified as Undetected, we can predict that five faults would be classified as Undetected in the Fault Space analysis. For example, if 'DFF1' in Figure 4.4 has a final weight of nine, and the fault classification of a fault injected in 'DFF1' output is Detected, we can estimate that the nine faults weighted in 'DFF1' would also be annotated as Detected. By repeating this analysis to all prime propagation nodes, we can estimate all faults in the circuit. Finally, by computing the total number of faults for each fault classification category, we can predict the result of the Fault Space analysis; this result also enables the estimation of the DC, and consequently, of the safety metrics.

4.2. VALIDATION AND RESULTS

The validation of the proposed methodology consists of a comparison between estimated fault classification results and actual fault injection results on target designs. By incorporating the actual results, we can analyze the estimation error rate and the performance improvements achieved by the methodology. Furthermore, the validation considered circuits with different physical characteristics and multiple simulation workloads, improving the overall assurance of the methodology efficiency. The AC97 is an audio codec controller IP compatible with a wishbone bus; it includes a functional testbench that verifies all circuit functionalities and a post-production test simulation environment based on ATPG [31]. The Conmax is an interconnect matrix IP core featuring a parameterized priority-based arbiter, including a functional testbench [31]. Finally, contemplating a test case representative from the automotive sector, we deploy the estimation methodology to the AutoSoC. The following sections detail the validation of our method and its deployment during the safety lifecycle; additionally, we demonstrate how to examine the computed parameters to extract an in-depth mapping of the design under the influence of faults.

The estimation of the Fault Space analysis results follows the phases described in 4.1

for each of the test cases. However, in an actual product lifecycle, the availability of the design abstraction level and simulation environment will vary according to the development stage. For that reason, the methodology supports the deployment of each analysis phase with the RTL or gate-level design descriptions. Furthermore, we can merge results from different abstraction levels by correlating the prime nodes; therefore, as the development advances, we can improve the estimation results by including more accurate design descriptions. Figure 4.5 illustrates the applicability of the proposed methodology in the different steps of the safety lifecycle; additionally, it highlights the hardware abstraction level we can deploy for the estimation on each phase.

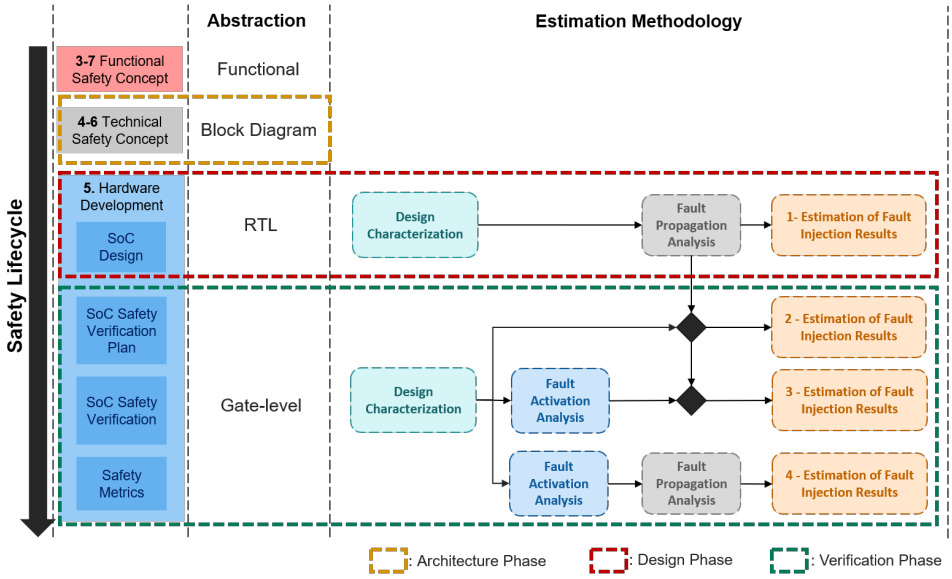

Figure 4.5: Application of the methodology during the Safety lifecycle

Generally, at the *5. Product development at the hardware level*, the design description is initially available at RTL; after verification of the implemented functionalities, the description of the hardware is synthesized into gate-level. It is also common to reuse the functional simulation environment for both abstraction levels; with such an approach, we can deploy the same testbenches and workloads to simulate the designs in RTL and gate-level. For that reason, the proposed methodology was conceived so each analysis phase can be performed on both RTL and gate-level, allowing the estimation of the safety metrics at different stages of the development lifecycle. Furthermore, initial estimates based only on RTL can be enhanced as the development advances. Figure 4.5 illustrated the options for deploying the estimation methodology during the safety lifecycle. The orange rectangles, named *Estimation of Fault Injection Results*, are numbered from one to four, representing the estimation results in the different development phases.

Initially, when the hardware description and simulation environment are only available in RTL, each phase of the proposed methodology should be performed at the abstraction level mentioned above; scenario *1 - Estimation of Fault Injection Results* in Fig-

ure 4.5. It is essential to notice that due to the characteristics of RTL, we could not see benefits from the Fault Activation Analysis phase; for that reason, we can bypass the execution of this phase in the analysis regarding only RTL. Next, when the initial versions of the gate-level are available, we can re-compute the results by only executing the Design Characterization; this scenario is highlighted as *2 - Estimation of Fault Injection Results*. In this case, we can reuse the RTL Fault Propagation Analysis results by correlating the prime propagation nodes from RTL and gate-level; the gate-level synthesis must include options to keep design hierarchy and terminology. For example, suppose the name of the sequential elements is the same in both abstraction levels. In that case, we can correlate the propagation, simulated in RTL, with the weights, from the gate-level, improving the estimation accuracy and avoiding the computational efforts from FI simulation at the gate-level. After, when the workloads are available for gate-level simulation, we can integrate the results from the Fault Activation Analysis into the previous results, as illustrated in *3 - Estimation of Fault Injection Results*. As fault activation has an essential role in the faulty gate-level behavior, calculating new weights based on this analysis improves the estimation results. Finally, scenario *4 - Estimation of Fault Injection Results* represents the final step of the methodology when all phases are executed with the gate-level resources; this is an extra estimation option to assess modification to the design before the final Fault Space analysis is conducted.

4.2.1. VALIDATION

Initially, the methodology was validated using the gate-level version of the more uncomplicated test cases; this approach enables the manual verification of the results, assuring that each flow step is operating as expected. In addition, it allows careful confirmation of the contribution of each phase to the estimation accuracy. Furthermore, we have configured the FI simulator to deploy the analysis using single strobes. In such a configuration, the faults will be annotated only as Detected or Undetected, facilitating the comparison between the actual FI results and the methodology estimation; it is crucial to notice that the methodology is applicable with any strobe configuration.

Figure 4.6 illustrates the results of the methodology applied to the AC97 test case; the top graph shows the results using a functional testbench, while the bottom presents the analysis with the ATPG testbench. The column **"FI Campaign"** shows the results of the FI simulation campaign; the other columns demonstrate the methodology results. From these, the **"Design Characterization + Fault Propag"** column shows the estimation results without Fault Activation analysis, and the **"Design Characterization + Fault Propag + Fault Activation"** column illustrates the final estimation results, deploying all methodology phases. The difference between the number of Detected in **"FI Campaign"** and **"Design Characterization + Fault Propag + Fault Activation"** demonstrates the error of the estimated results. When simulating the functional testbench, the methodology computes 42.166 Detected faults, while the FI Campaign identifies 39.863 faults in such a category; additionally, this test case highlights the importance of the *Fault Activation Analysis* to improve the estimation results. By excluding the weights of faults that are not activated by the workload, the Detection estimation decreased from 49.172 to 42.166 faults, achieving a difference of 4% between estimated and actual results.

Next, the bottom graph of Figure 4.6 illustrates the results when deploying the AC97

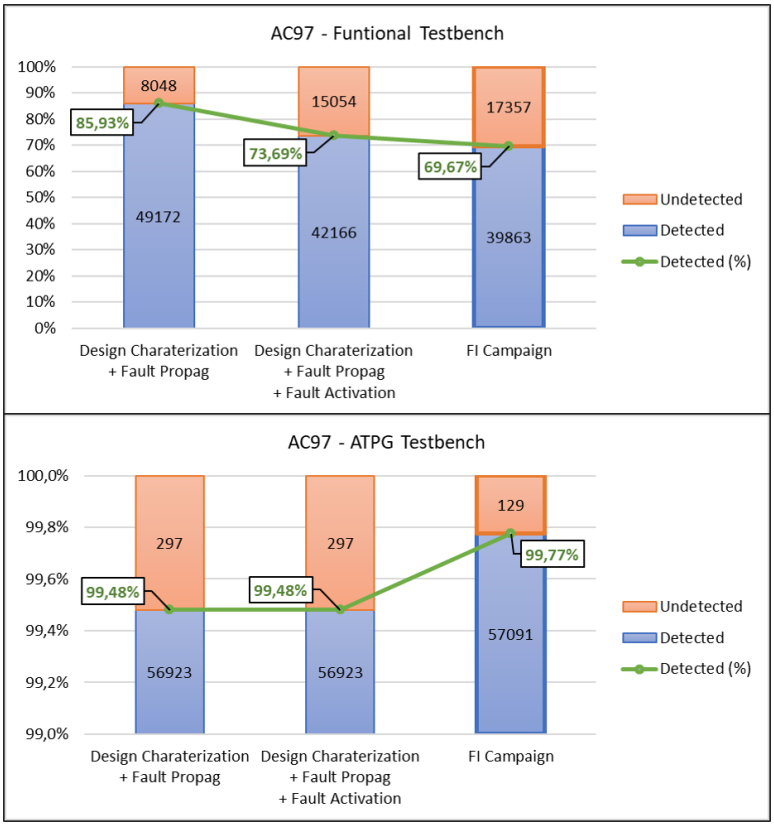

Figure 4.6: AC97 with Functional and ATPG Testbenches Analysis Result.

test case with the ATPG testbench. As both estimation and FI results are above 99%, the graph's vertical axis shows the results from 99% to 100%. This example highlights the impact of the workload for fault classification; even though the hardware design is the same, the behavior of the faults is entirely different when applying ATPG stimuli. Another essential observation is that the methodology result is the same with and without Fault Activation analysis; as the test stimulus was generated by an ATPG tool, the workload enables the activation of all faults, balancing the benefits of this analysis. In this example, with almost 100% detection, the estimation error is due to the weight computation during the design characterization. Even though the weight rounding results in a maximum estimation of 56.923 Detected faults, the difference between estimated and actual results in the example is minimal.

Finally, we conclude the validation with the analysis of the Conmax test case using a functional testbench. The Conmax implements a control matrix, enforcing parallel processing, differently from the previous test case; also, the design has a higher complexity, with an almost three times larger Fault Space. Figure 4.7 illustrates the estimation results. Likewise, we can see the benefits of the Fault Activation analysis in the middle column;

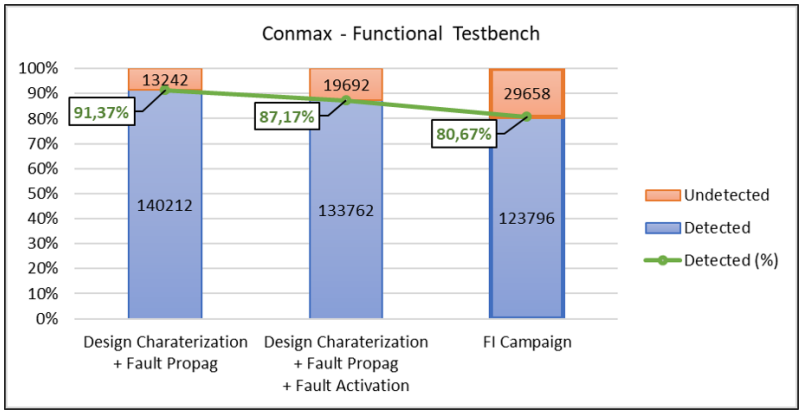

Figure 4.7: Conmax with Functional Testbench Analysis Result

after removing 6.450 faults that are not activated, the Detection estimation rate is 87,1%, resulting in a difference of 6,4% when comparing with the FI Campaign results.

4.2.2. SAFETY LIFECYCLE RESULTS

After the validation with more straightforward test cases, where we could manually verify the results and validate the methodology, it was necessary to prove it with a representative design comprising the resources for mimicking an actual safety lifecycle. As illustrated in Figure 4.5, for such demonstration, we need the hardware described in multiple abstraction levels, a compatible simulation environment, and various workloads. Considering these requirements, we select the AutoSoC as the target for the design safety metrics estimation. This section describes the results of the estimation methodology applied to the AutoSoC using three different software applications; also, it follows the flow demonstrated in Figure 4.5. First, we execute all analysis phases using the RTL abstraction level. Then, as gate-level becomes available, the RTL results are merged with the design characterization at the gate-level. Next, we compute the Fault Activation analysis, improving the estimation results. Finally, all methodology phases are deployed at the gate-level, generating a final estimation before an entire FI campaign is required.

Figure 4.8 illustrates the analysis results at RTL; this step is represented by *1 - Estimation of Fault Injection Results* in Figure 4.5. The analysis comprises three software applications available in the AutoSoC benchmark: Hello World, a bare metal hello world application; CalcPrime, an application based on RTEMS operational system that verifies if a range of input values are prime numbers; STL, a Software Test Library deploying a sequence of opcodes to propagate faults to the CPU outputs.

The methodology demonstrates a good estimation in the RTL abstraction level; estimation results have a difference between 1,5% to 2,3% in the Detection rate compared to the FI Campaign results. However, it is essential to consider that the Fault Space in RTL is different from the gate-level; even though the FI results in RTL indicate the design behavior under faults, it is necessary to repeat the analysis at the gate-level for a precise classification. For that reason, as the design development progress, we can improve

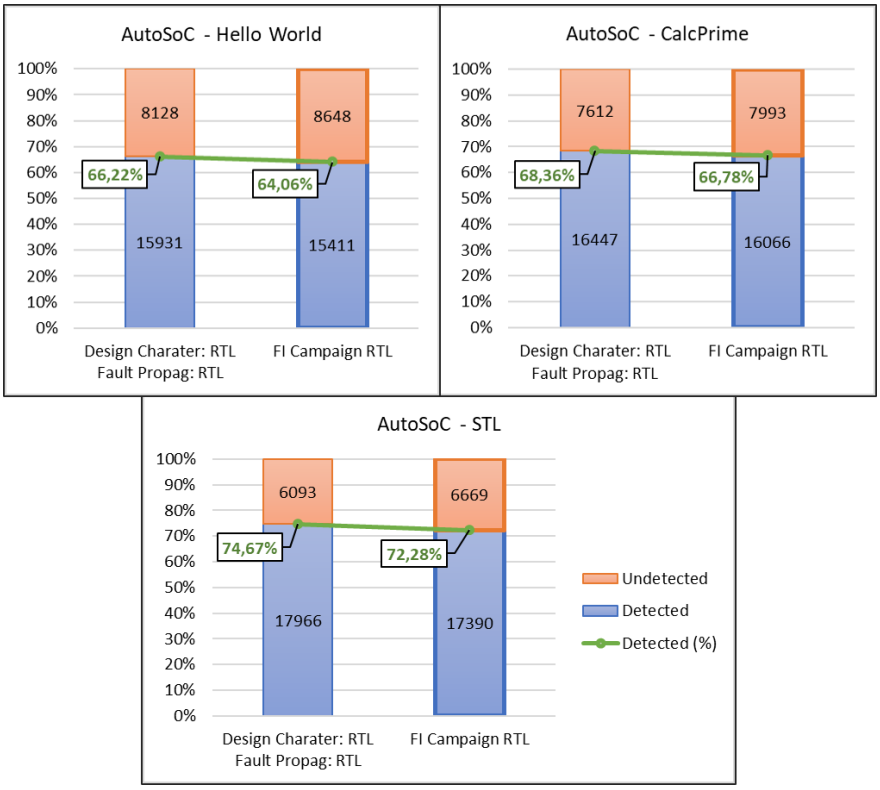

Figure 4.8: AutoSoC RTL Analysis Result

the estimation by repeating phases of our methodology at the more accurate abstraction level.

The following Figures illustrate the results of the *Estimation of Fault Injection Results* steps two to four according to Figure 4.5. Each Figure represents the analysis results for one of the software applications. For example, figure 4.9 illustrates the results when executing the HelloWorld; Figure 4.10 shows the CalcPrime estimation results; Figure 4.11 displays the STL analysis output.

The first example, in Figure 4.9, illustrates the analysis results when simulating the Hello World application in the AutoSoC; as this is a simple application, generally being used as an initial functionality test for compilers and CPUs, it provides a low fault propagation rate. This behavior is also observed in the estimation results. The estimation of Detected faults starts with 44.672 when applying the *2 - Estimation of Fault Injection Results* flow; next, when deploying Fault Activation, the result is 40.729 Detected faults; and finally, when executing all phases with the gate-level, the estimation of detection is 41.110. As the actual number of Detected faults is 38.964, the methodology provides a detection rate estimation with an error varying from 5,9% to 1,8%.

Next, as illustrated in Figure 4.10, we deploy the CalcPrime workload. Such an ex-

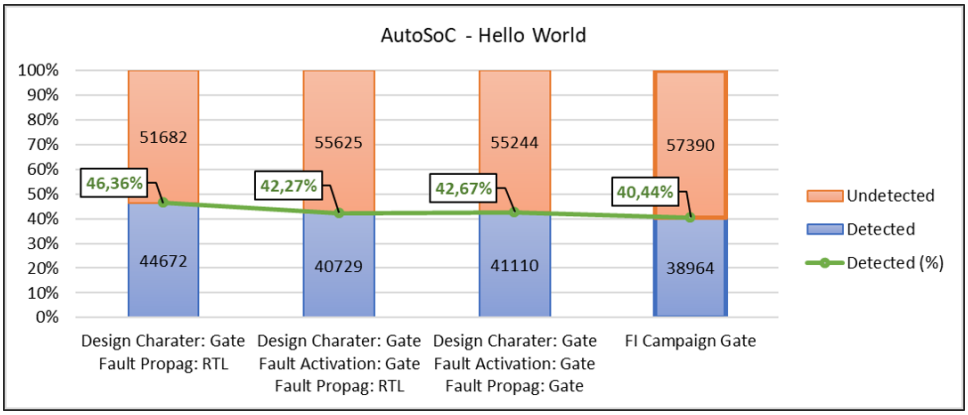

Figure 4.9: AutoSoC Hello World Analysis

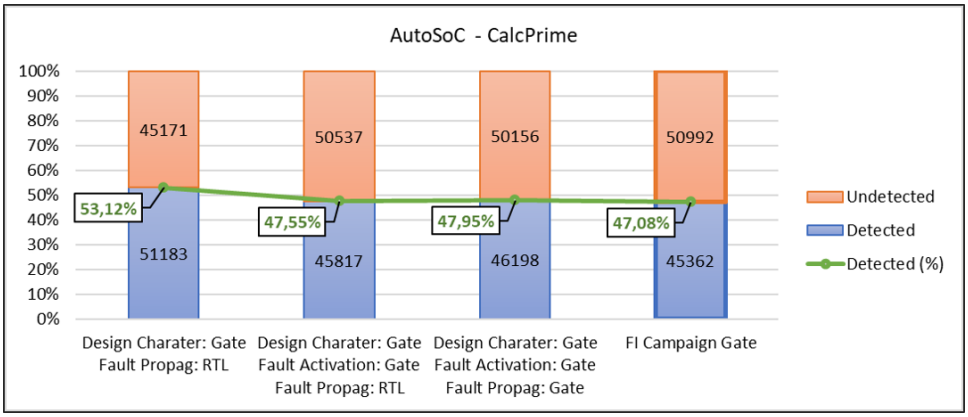

Figure 4.10: AutoSoC CalcPrime Analysis

ample provides a higher activation of the CPU blocks when compared to the previous one; the calculation of prime numbers requires the arithmetic unit's usage and general-purpose registers for storing partial values. As a result, the workload exercises more CPU components, providing better fault propagation rates. In this example, the methodology estimates 51.183, 45.817, and 46.198 Detected faults for each analysis phase. Meanwhile, the FI Campaign resulted in 45.362 Detected faults, representing a detection rate of 47%. Therefore, the difference between estimated and actual starts with 6% in *2 - Estimation of Fault Injection Results*, and achieves 0,4% and 0,8% in the following phases.

Finally, Figure 4.11 shows the methodology results when simulating the STL application on the AutoSoC. An STL is a collection of software tests targeting faults' propagation or detection; it corresponds to a set of software procedures, usually developed in assembly code, C code, or both. As these applications are specifically designed to propagate the faults in the CPU, they achieve higher detection rates when compared to the previous workloads. Likewise, the results supplied by the estimation methodology

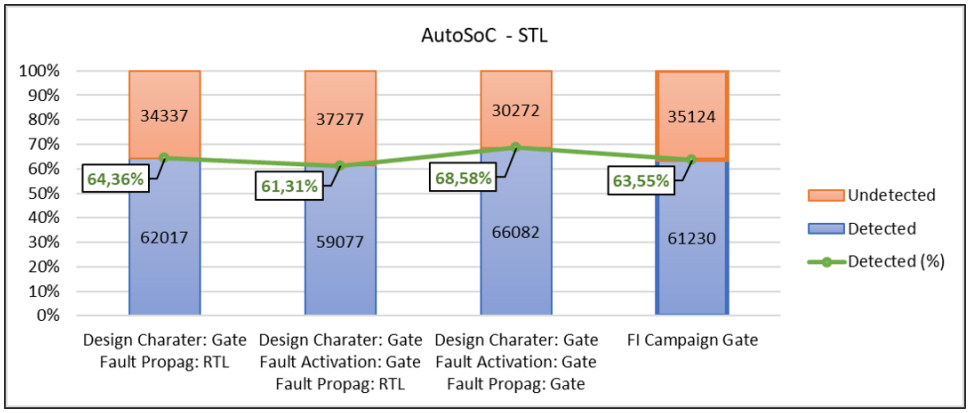

Figure 4.11: AutoSoC STL Analysis

are in similar figures. For example, the actual FI Campaign results report 61.230 faults as Detected. Meanwhile, the methodology phases estimated 62.017, 59.077, and 66.082 Detected faults, respectively. Therefore, in this example, our estimation achieved a difference between 0,8% and 5% compared to the actual Detection rates.

Unlike the other test cases, the measure of phase *2 - Estimation of Fault Injection Results* is the more accurate; however, we need to understand the reasons for such a discrepancy. In test cases like STL, aiming to exercise particular hardware components, our methodology tends to have a higher difference from actual results; the design characterization calculates the weights as a representation of the Fault Space. Therefore, as with any representation model, there is a natural loss of accuracy compared to the actual Fault Space; consequently, hardware components targeted by the STL may not be represented in the design characterization model, increasing the estimation error. Considering that the RTL contains a subsampling of the gate-level Fault Space and that the STL targets specific gate-level components, the RTL-based estimation's higher accuracy results from elements not represented in this abstraction level. In such a scenario, the RTL's smaller Fault Space reduces the number of faults estimated as Detected, decreasing the difference to the actual FI Campaign results. Nevertheless, the estimation results provided by the methodology are in a good accuracy range enabling early estimation of the safety metrics even in test cases like the STL.

4.2.3. ADDITIONAL DESIGN EVALUATION

As previously discussed, the workload has a crucial function in the activation and propagation of faults; a fault can only be declared as Detected if the test stimulus applied to the circuit promotes the activation of the faulty element and enables the propagation of the fault effect to the strobes. Therefore, for assessing the workload importance, the proposed methodology computes its contribution, recalculating weights during Fault Activation analysis and evaluating their influence in the Fault Propagation analysis. The results above demonstrate that the cumulative information extracted by the design characterization and workload analysis provides an accurate model for estimating fault de-

tection. Furthermore, by studying this information, we can extract an in-depth mapping of the design under the influence of faults; such an analysis demonstrates crucial details about fault propagation potential and critical areas of the circuit. Table 4.1 details the information extracted during the analysis of the AutoSoC.

Table 4.1: Detailed AutoSoC Weighting Analysis

AutoSoC Analysis (SA0/SA1)	Prime Nodes	Total Weight	Max Weight	Average Weight
Design Characterization	4648	83427	1806,4	33,8
Design+Workload Hello World	4648	66240	1232,2	26,8
Design+Workload CalcPrime	4648	67314	1301,9	27,3
Design+Workload STL	4648	76110	1806,4	30,8

First, the row *Design Characterization* demonstrates the results without analyzing the workloads. The column *Total Weight* demonstrates the numbers computed during the design characterization for the AutoSoC. After the design characterization, the weight represents the fault propagation potential of the circuit; if a workload could activate and propagate all faults in the design, the *Total Weight* would describe the number of detected faults. The column *Max Weight* promotes the identification of critical nodes in the design, denoting the maximum weight in a single node. The column *Average Weight* shows the *Total Weight* divided by the number of *Prime Nodes*; representing the distribution of fault nodes over the sequential elements in the circuit.

Next, each additional row demonstrates the results after analyzing a given workload; if the workload does not activate a fault, the weight is recalculated accordingly. Therefore, the *Design+Workload* rows represent the maximum fault detection coverage when simulating the given workload. Furthermore, we can deploy the *Total Weight* after the workload analysis to rank the workloads by their potential for fault detection. The workload with resulting parameters closest to the netlist characterization parameters will likely have higher fault detection rates.

By analyzing Table 4.1, we can conclude that the STL workload has a higher potential for fault propagation. First, the *Total Weight* is higher than the other workloads, meaning that the STL activates more faults. Also, the *Max Weight* value is the same before and after the STL analysis, indicating that this workload could exercise all faults in the most critical area of the design. Finally, the difference between the *Total Weight* of the *Design Characterization* and the *Design+Workload STL* rows reveals the number of faults that the given workload cannot detect; highlighting a gap for improvements in the fault coverage.

4.2.4. SUMMARY OF RESULTS AND DISCUSSION

As previously described, the validation of the results consisted of comparing the actual results of the Fault Space analysis with the estimation of Detected faults for each test case. For such, we configured FI Campaigns to simulate SA0 and SA1 faults on every cell port of the circuits. Table 4.2 summarizes the result considering the gate-level descrip-

tion of the test cases. The columns labeled as *Actual Detected* represent the results from the FI Simulation; the columns labeled as *Estimated Detected* outline the results from the proposed methodology; the *Estimation Error* column shows the difference between Estimated and Actual; and, finally, the column *Exec. Time Improvement* highlights the performance gain when comparing the estimation method with the actual FI Campaign.

Table 4.2: Results Summary

Design	Workload	Total Faults SA0/SA1	Actual Detected	Actual Detected (%)	Estimated Detected	Estimated Detected (%)	Estimation Error	Exec. Time Improvement
ac97	Funct TB	57220	39863	69,67%	42166	73,69%	4,02%	8,5X
	ATPG TB	57220	57091	99,77%	56923	99,48%	-0,29%	5,8X
conmax	Funct TB	153454	123796	80,67%	133762	87,17%	6,49%	12,8X
autoSoC	helloWorld	96354	38964	40,44%	41109	42,66%	2,23%	17,8X
	calcPrime	96354	45362	47,08%	46198	47,95%	0,87%	19,1X
	STL	96354	61230	63,55%	67083	69,62%	6,07%	20,4X

The execution time of the FI Simulation campaigns depends on several factors. However, the most critical parameter is the availability of resources for executing parallel simulations. For the designs with a faster simulation time (e.g., the AC97 and the Conmax), the FI Simulation campaign is configured sequentially, resulting in execution time in the granularity of days to a week. For the AutoSoC, which contemplates the simulation of an entire SoC, the FI simulation is configured in concurrent mode, executing up to 100 faults in parallel, resulting in execution times in the granularity of weeks to a month; the estimation technique execution time is faster than the FI Simulation campaigns in all cases. Table 4.2 highlights the main achievements from the proposed methodology:

- Accuracy: The column "Estimation Error" highlights the difference between the Estimated fault detection rates and the Actual fault detection;
- Efficiency: The column "Exec. Time Improvement" notes the performance gain when deploying the proposed methodology.

As illustrated in Table 4.2, the results achieved by the proposed methodology are encouraging; it estimates fault detection rates with an accuracy between 0,2% and 6,4%, with up to 20X faster execution times. These figures allow for a confident design space exploration of a hardware design concerning the safety metrics. By deploying such a technique, safety engineers can explore diverse architecture possibilities with a higher degree of certainty. It is crucial to note that the proposed methodology cannot substitute the execution of the Fault Space analysis as part of the Functional Safety requirements of ISO 26262. However, even though the hardware design must undergo Functional Safety verification at the final stages of development, the feasibility of an early estimation of the safety metrics sustains safety-related architectural decisions.

4.3. CONCLUSIONS

Functional safety verification is a critical step for ISO26262 compliance. At later stages of safety-critical systems development, designers must analyze the behavior of the design under the effect of faults to show conformity with the expected safety metrics. Failing to achieve these conditions entails additional iterations through critical development and

verification phases. This chapter presents a methodology for the design space exploration of safety architectures. By allowing engineers to estimate safety metrics at earlier development stages, we provide a tool for investigating safety architectures, improving the confidence in conceptual decisions, and decreasing the chances of rework. We tackle this issue by allowing the estimation of Detected faults throughout the safety lifecycle. We consider the current hardware abstraction level for each development stage to model faulty behavior; as the development progresses, we enhance the results by computing more accurate hardware descriptions. Our results demonstrate the accuracy of the technique by providing an estimation of the fault detection rate with an average error of 3%. Moreover, the methodology results in an execution time up to 20X faster when compared with the traditional gate-level Fault Injection campaigns.

5

ENHANCING ONLINE FAULT DETECTION OF AUTOMOTIVE CPUS

The advances in Automotive applications increase the demand for safety solutions targeting online fault detection with low-cost overhead. Unfortunately, DfT is often not an option during operational life; redundancy methods such as the DCLS may be prohibitive due to the overhead. In contrast, STLs appear as a cost-effective alternative for the online detection of random faults, but the development efforts are usually a problem. This chapter proposes a formal-based technique for generating STLs. First, we constrain the formal environment to use only pre-selected CPU instructions; then, the formal verification will determine a sequence of such instructions to propagate a given fault. Furthermore, we modify the traditional strobes configuration to calculate a signature indicating the presence of faults; such an approach results in a standalone safety mechanism enabling detection of faults. Finally, the technique is validated using the AutoSoC CPU; the automatically generated STL achieves a detection rate of 53% for SA0/1 faults in the CPU digital area.

THE increasing complexity in automotive applications is causing a shift in the traditional design flow. For instance, an Integrated Circuit (IC) that implements safety-critical applications, such as autonomous driving, must incorporate mechanisms to reduce the risk of failures resulting in life-threatening situations. For such applications, the system must detect an extremely high percentage of potential faults while already deployed in the field. This process becomes challenging in the most advanced automotive ICs, where millions of design components are susceptible to random hardware faults. Furthermore, the demands for fault detection during the design operation require the deployment of suitable test mechanisms; Design for Testability (DfT) is often not an option, as it could disturb the intended functionalities. Another common approach in safety-critical domains is to deploy redundancy of systems and components, such as the Triple Modular Redundancy (TMR) or Dual Core Lock Step (DCLS). However, the overhead costs of redundancy schemes have a substantial impact on automobiles; the consequence on the final price may be prohibitive in such a competitive market. The growing demand for Automotive applications, together with functional safety requirements from ISO 26262, demands innovative methodologies to increase operational safety without significant increases in the cost.

The typical approach for developing automotive CPUs compatible with ASIL D is the DCLS. However, not all applications require the highest integrity levels, and many cannot afford the additional cost; a tradeoff between costs and safety would suffice for many automotive systems. Consequently, safety schemes based on Software-Based Self-Testing (SBST) are a good alternative for achieving high integrity levels without the overhead of redundant hardware. The purpose of such methods is to achieve high fault detection rates without imposing excessive overhead in the test budget [52]. Furthermore, solutions based on Self-Test Library (STL) - a collection of Software-Based Self-Test (SBST) procedures - are widely deployed for the verification of CPUs [68][53][58].

Nonetheless, not all approaches are suitable for Automotive applications. For example, most proposed STLs target the propagation of faults to the outputs of the circuit, similar to the ATPG method; for these to work, it would be necessary to include components to monitor the design outputs and identify the presence of faults. The approach based on STLs with standalone capabilities for fault detection is primarily adopted in the automotive industry [69][70][71]. However, the manual development of such STLs is very complex, demanding specialized resources and long development cycles.

This chapter proposes a technique that enables the automatic generation of STLs, targeting online fault detection in automotive CPUs. For such, we explore the strengths of formal methods in determining optimal test stimuli for fault propagation to create a sequence of commands in the format of an STL. First, the process deploys Formal Tools to analyze the propagation of faults in a CPU; for such, we create a set of properties to enable the formal environment to use only pre-selected opcodes for fault propagation. Next, we extract the sequence of opcodes deployed to propagate the fault, also known as the counter-example of the formal property. Then, by integrating all opcode sequences, we generate an STL capable of detecting faults in the CPU. Finally, the technique is validated using the AutoSoC Bechnmark as a test case. The results of the FI campaign, using the generated STL, show a standalone detection rate of 53% for SA0/1 faults in the CPU digital area.

5.1. FORMAL PROPERTIES AND COUNTER-EXAMPLES

As discussed in 1.3.1, formal analysis is a powerful technology to understand the behavior of a circuit. For example, such tools can determine the testability of faults by verifying the design's physical characteristics. Also, by generating a representation of the design's boolean function, they can determine the validity of formal properties. The analysis of the properties must consider every possible combination of input values; a property is True if the formal engine identifies a sequence of test stimuli that activate it. Figure 5.1 illustrates a simple circuit example with inputs (in), outputs (out), gates (g), and a fault target (f).

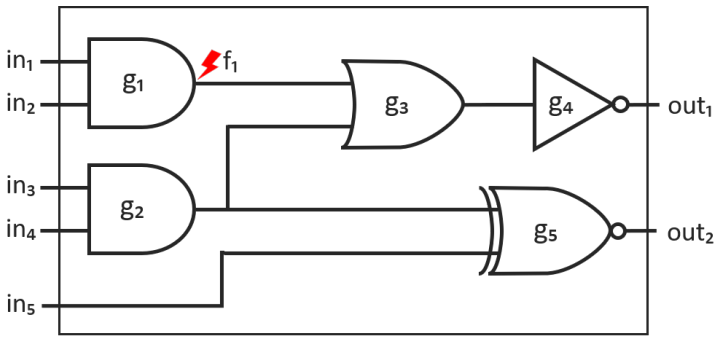

Figure 5.1: Circuit Example for Formal Analysis

The example illustrates the analysis of a fault target 'f1' in the output port of 'g1'; in the context of fault analysis, the formal tool would verify activation and the propagation of 'f1' to one of the circuit's outputs. For example, the activation analysis of SA0 and SA1 faults would generate the following properties:

$$\text{SA0 activation property: } assert\ property\ (g1.out == 1) \tag{5.1}$$

$$\text{SA1 activation property: } assert\ property\ (g1.out == 0) \tag{5.2}$$

Then, the formal engine would apply all possible combinations of values to the inputs (in1 to in5) and confirm the cases where the properties are True. In this simplified example, the value of 'f1' depends only on the test stimulus applied to 'in1' and 'in2'. Therefore, the engine would determine that property 5.1 is True when 'in1' and 'in2' receive the logic value one; and property 5.2 is True when 'in1' or 'in2' acquire the logic value zero. The combination of test stimuli that induces a property to be True is named the counter-example of the property. Still considering the given circuit, if the activation properties are True, the formal engine would also perform the propagation analysis; it would generate properties to demonstrate if a value applied to 'f1' can reach one of the outputs. Likewise, the formal engine would verify if the propagation properties are True, considering all combinations of test stimuli; if they are, we could extract the counter-example of the properties, meaning the input values that drive fault activation and propagation. Furthermore, the counter-example of a given property can be deployed in FI Simulation to reproduce the behavior.

The formal properties validation using the counter-example is a well-established method for verifying hardware designs; it is a powerful resource for debugging the design behavior. In the context of fault space analysis, it is also an effective option to generate test cases with sound fault propagation rates. Another significant advantage is that state-of-the-art tools can automatically generate formal properties to classify faults, reducing formal verification environments development efforts. Furthermore, these tools provide features to help us understand the behavior changes caused by faults. For example, Cadence® JG FSV includes the fault detection trace utility; it provides an advanced analysis window highlighting the circuit state, sequence of test stimuli, and the number of cycles that enable the effect of the fault to reach the design outputs. Next, we will describe an example of the fault detection trace utility applied to the fault analysis of a CAN controller. Figure 5.2 illustrates the example test case, highlighting the inputs and outputs of an open hardware implementation of the SJA1000 CAN Controller, developed by Philips in the early 2000s.

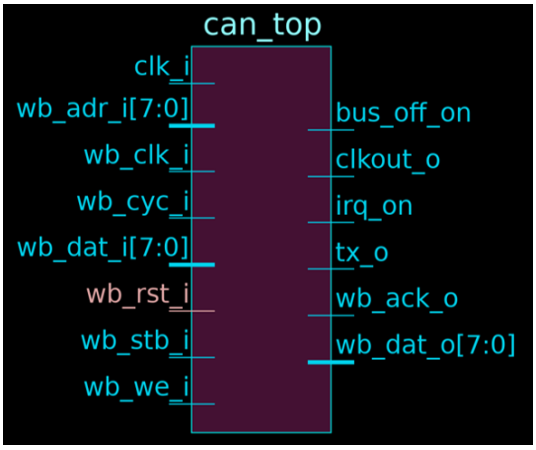

Figure 5.2: CAN Controller Top Level

The CAN controller is compatible with the Wishbone bus; therefore, most circuit stimuli are supplied by the bus's connections; these are the input ports named with the "wb_" prefix, representing the main source of stimulation for propagating the circuit faults. Additionally, the circuit includes an external clock. In the example, the fault space analysis will examine SA0 and SA1 faults in every cell port; for each fault, the formal engine will generate properties to show activation and propagation; the proofing of such properties will consider every combination of input values. For each fault classified as Detected, we can deploy the detection trace utility to verify the conditions that cause fault propagation. Figure 5.3 demonstrates the fault detection trace when analyzing an SA0 fault in the *addr[1]* signal of the CAN Controller.

The Figure details the circuit signals that are relevant for the propagation of the fault under analysis; the signals are always displayed in pairs, the value in the Design Under Test (DUT), and the counterpart in the Bad Machine. As described in 1.3.1, the Bad Machine is a copy of the DUT, where the faults are injected; the formal engine connects the

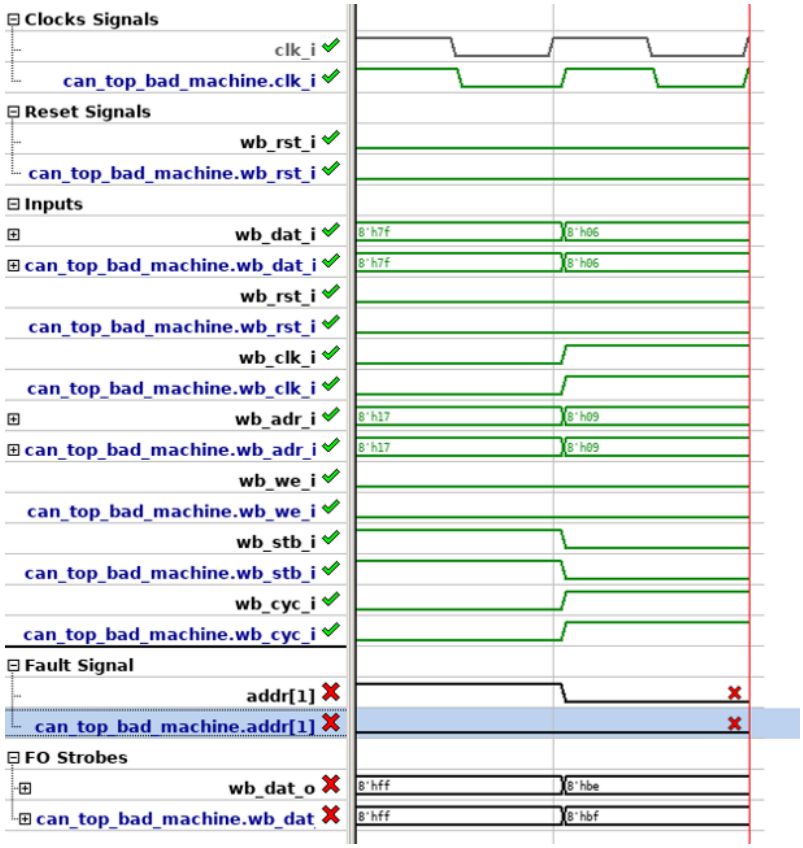

Figure 5.3: Fault Detection Trace

inputs of both and monitors the outputs; any discrepancy in the behavior of the machines indicates the fault effect. Furthermore, the user must indicate the signals responsible for the clock and reset sequences of the circuit; these configurations are necessary so the formal tool can interpret the timing and design's initialization state. In Figure 5.3, the control signals previously configured in the tool are shown under **Clock Signals** and **Reset Signals**. The example illustrates the SA0 fault under **Fault Signal**; we can see that *can_top_bad_machine.addr[1]* maintains a logic value of zero, while *addr[1]* is initiated with a logic value of one. Additionally, the group **FO Strobes** highlights the propagation of the fault to the circuit output *wb_dat_o*; after one clock cycle, we can already see a discrepancy between the DUT and the Bad Machine outputs. The test stimuli that influence the propagation of the SA0 fault in *can_top_bad_machine.addr[1]* to the output *can_top_bad_machine.wb_dat_o* is detailed under the group **Inputs**. These are the values applied by the formal engine to the circuit's inputs to propagate the fault; in other words, this combination of values is the counter-example that the activation and propagation properties are True. Finally, considering that the user-defined clock and reset configuration are accurate, we can reproduce the sequence of values from **Inputs** in a

testbench, confirming the propagation of the fault in the FI simulation environment.

The strength of formal verification in identifying scenarios that confirm the design's properties is crucial for analyzing the fault space. This technology is not limited to a specific time or state; instead, it contemplates a global scope, considering every evaluation context and test stimuli. Consequently, formal methods can exhaustively prove that a fault can never produce any failure; in cases where no existing combination of test stimuli can propagate it. In the opportunities where an assortment of test stimuli causes fault propagation, the tool can identify such values as the counter-example of the formal properties. Furthermore, the method applied by such technology for verifying properties is the foundation for enabling the automatic generation of the test libraries.

5.2. AUTOMATIC GENERATION OF SOFTWARE TEST LIBRARIES

As previously described, solutions based on STLs deploy a collection of Software-Based Self-Test (SBST) procedures for detecting faults during the operational phase of CPUs. The concept is similar to the CAN controller example previously discussed, where a combination of inputs is applied to the design to identify the presence of faults. The difference is that in the case of CPUs, which are mainly controlled by software instructions (or opcodes), a sequence of such instructions is the primary source for identifying the presence of faults. Furthermore, we explore the formal capability for generating counter-examples of fault propagation to create a sequence of commands in a STL format.

When deploying the fault space analysis of a CPU using formal methods, we must consider specific characteristics of such design. First, we must configure the control signals of the CPU in the formal environment; the engine must comprehend complex clock threes and reset sequences. Also, CPUs tend to include several debug features; these additional inputs and outputs may not be relevant during its operational phase and should be configured accordingly. Likewise, we must inform the tools about external signals responsible for stopping the execution of the CPU (e.g., interruptions, JTAG, and so on). Finally, we need to identify the main inputs and outputs for fault propagation; these will generally be the connections to the instruction and data memory; they can also be other connections to application-specific peripherals. Figure 5.4 illustrates an example CPU; highlighted in blue are the control signals; and, marked in red, the inputs from the instructions and data memories.

One of the advantages of formal verification is that the analysis doesn't require the presence of testbenches; the technology deploys every combination of inputs, not limited to the scenarios conceived in a simulation environment. However, for complex circuits like CPUs, not all combinations are usually relevant; considering the design surroundings, we can decrease the level of freedom of the formal analysis, improving the results. If, for example, a given SoC that includes the example CPU doesn't incorporate peripherals to the ports **others_in** and **others_out**, we can configure this information in the formal engine with *assume* statements. Furthermore, by decreasing the exploration space, we facilitate the proving of formal properties, improving execution times. Equation 5.3 describes an *assume* statement indicating to the verification environment that the connection **others_in** is open; it will always hold the logic value zero.

$$assume -env \{cpu\_example.others\_in[31:0] == 32'h00000000\} \tag{5.3}$$

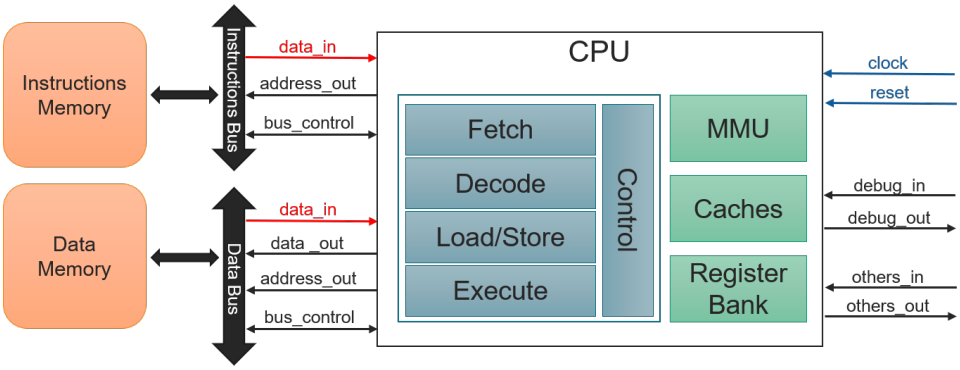

Figure 5.4: Example CPU

Another essential aspect to consider is the operating modes available in a CPU. In general, the execution control of a CPU includes several flows, initialization of internal blocks, handling of interruptions and exceptions, external control via debuggers and JTAGs, among others. Furthermore, not all combinations of test stimuli are consistent with all operating modes. For that reason, setting up the proper operational mode is crucial to validate the formal verification results. For such, we can include *assume* statements replicating the external inputs that set the CPU into the desired mode. For the specific case of the STLs, we want to perform fault analysis in operational mode; as such, we must configure the formal environment accordingly. In the example CPU illustrated in Figure 5.4, the setup would include proper configuration of clock and reset signals, assuring proper initialization of all internal blocks; and the inclusion of *assume* statements in the **debug_in** input to disable external CPU control.

Next, we need to identify the operational outputs where we would like to analyze fault propagation. As previously discussed, we determine the behavior change provoked by a fault when the effect is observable in one of the outputs (strobes); the test stimuli are responsible for triggering such propagation to the strobes. For that reason, the classification of faults relies not only on the test inputs but also on the outputs configured as strobes. Furthermore, when assessing Automotive designs, we must consider ISO 26262 definition for the fault space classification; a fault is dangerous if it can violate a safety goal. Therefore, the configuration of strobes in the example CPU should include only outputs that can affect the system's functionalities. In Figure 5.4, we could determine **address_out** to both buses and **data_out** to the Data Bus as the operational outputs; an error propagate to these outputs could affect safety-critical functionalities. The output **others_out** could also be considered depending on the function implemented by the peripherals connected to it; the **debug_out** could be ignored as the propagation of errors to this output are not relevant for critical functions.

Finally, the formal engine will generate properties that demonstrate fault propagation to the configured strobes; the validation of such properties will consider every possible combination of values in the inputs of the CPU; in the example, these inputs are the **data_in** from both buses. By default, the verification environment has no previous knowledge regarding the format of the data inputs; however, in the case of the **data_in**

from the instructions memory, the structure of the information should observe the CPU's Instruction Set Architecture (ISA). Furthermore, we want to assure that the formal properties are confirmed using proper instructions as test stimuli; as a result, the counter-examples can be extracted and converted back to assembly opcodes; next, the sequence of opcodes is formated as a software function.

The input from the instructions memory to the CPU follows a numeric machine code format. The ISA defines such a code, including the functionality to be executed by the CPU and the required internal registers. In addition, the ISA describes the opcode format, allowing compilers to translate it to machine code. For instance, Table 5.1 describes an addition operation on the OpenRISC architecture. The Table also highlights the mathematical function to be executed, the instruction format defined in the ISA, the assembly command, the machine code in hexadecimal and binary formats, the size of each field, and the bitwise mapping in bits for each instruction operator.

Table 5.1: Example of a 32 bits CPU Instruction

CPU Functionality	$RegD = RegA + Value$			
Instruction Format	Opcode	Register D	Register A	Value
Assembly Example	l.addi	Reg1	Reg2	100
Hexadecimal Code	0x27	0x01	0x02	0x0064
32 Bits Instruction	10 0111	0 0001	0 0010	0000 0000 0110 0100
Field Size	6 bits	5 bits	5 bits	16 bits
Bitwise Map	31 - 26	25 - 21	20 - 16	15 - 0

The example in Table 5.1 describes a 32 bits instruction split into four fields: first, the *Opcode* defines the operation to be executed by the CPU; next, the *Registers D and A* specifies the CPU's general-purpose registers to hold the procedures partial and final values; lastly, *Value* appoints an auxiliary integer value. By default, the formal engine is not aware of the ISA definition; therefore, it will randomly apply test stimuli to the instructions **data_in** input until it concludes the verification of the properties. However, we can deploy *assume* statements to configure appropriate input conditions in the formal environment; by doing so, we potentialize the verification and assure the counter-examples from proven properties can be replicated as proper CPU instructions. For such, we would define only the *Opcode* field of the instruction as an *assume* statement; the formal engine has the freedom to determine the combination of the other fields to generate a complete instruction (or sequence of instructions) that causes fault propagation. The equation below describes an *assume* statement configuring the instruction illustrated in Table 5.1 in the formal environment; also, the example illustrates two restrictions on the *Register D* field, limiting the values that the formal engine can use. In the example, *Opcode* field (bits 31 to 26) is configured with the machine code corresponding to the *l.addi* instruction; the field *Register D* (bits 25 to 21) must comprise values different from 0 and 16, as writing to these registers could perturb the software execution.

$$assume -env \{ (cpu\_example.data\_in[31:16] == 6'b100111)\ and \\ (cpu\_example.data\_in[25:21] != 5'b00000)\ and \\ (cpu\_example.data\_in[25:21] != 5'b10000) \}$$

The counter-example of the formal verification of propagation properties, including the constraint above, would consist of a sequence of *l.addi* commands, with different values for the *Register D*, *Register A* and *Value* fields. As we are overconstraining the test stimuli space to a single instruction, the fault space analysis would result in fewer Detected faults. However, for the faults classified as Detected, we could extract a counter-example with direct translation into a sequence of assembly commands. Furthermore, we can define several opcodes that the formal engine can apply as stimuli; therefore, we can expand the CPU area exercised by the test stimuli increasing the coverage. By improving the number of opcodes available for the formal engine, we tend to have a higher number of Detected faults; as such, the verification of propagation properties will specify a sequence of pre-determined opcodes that will be the base of the STL.

5.3. CONFIGURATION FOR THE AUTOSOC

This work deploys the AutoSoC Benchmark as the test case to validate the previously discussed techniques. As described in 3, the AutoSoC includes several configurations targeting different ASILs; the *AutoSoC STL* configuration targets higher integrity levels without the overhead required by redundancy schemes as DCLS. For such, the STL must cover a good portion of the digital area of the CPU; combining the ECC in internal memories with a good STL coverage results in an inexpensive version of the SoC that complies with Automotive requirements. Figure 4.11 illustrates the AutoSoC STL configuration including the CPU primary inputs and outputs; the signals **ibus_ctrl** and **dbus_ctrl** symbolizes the several control signals of the Wishbone bus.

5.3.1. CONTROL SIGNALS AND OPERATIONAL MODE

Similar to the example CPU described in the previous section, we need to properly configure the formal environment to analyze the AutoSoC CPU. First, we need to set up the control signals; the mor1k implementation of the OpenRISC includes a simple clock three and synchronous reset; for that reason, a simple indication of the clock and reset signals suffice for the control configuration. Next, we must ensure that the formal engine performs the fault analysis considering the correct operational mode of the CPU. For such, we analyze the CPU simulation in operating mode and replicate the conditions applied by the testbench to the formal engine; this analysis results in a series of *assume* statements setting up the state of external and internal values of the CPU. With a concise definition of the signals that control the CPUs operating mode, the formal engine can analyze considering only conditions valid for STLs. Listing 5.1 exemplifies the assumptions applied to the AutoSoC verification environment to ensure the correct operational mode is considered. It includes the signal states of external inputs, such as interruption, debug unit, buses errors, and internal signals not used in the current CPU configuration. The

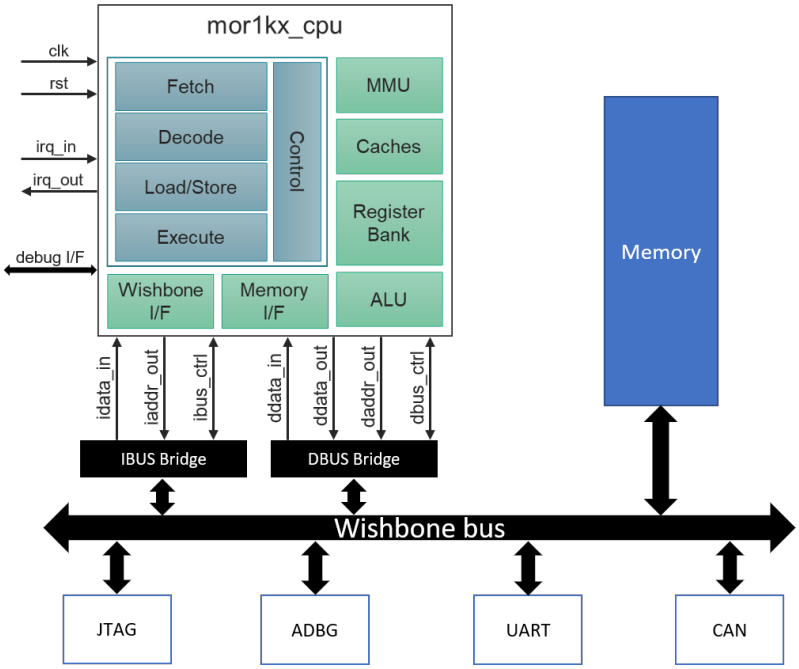

Figure 5.5: AutoSoC STL Configuration

configuration of internal signals values may not impact the operational mode but reduce the analysis exploration space, improving the results.

```
# Inputs from External Interruptions
assume -env {mor1kx_cpu.irq_i[31:0] == {32'h00000000}}
# Inputs from Data and Instruction Buses
assume -env {mor1kx_cpu.ibus_err_i == {1'b0}}
assume -env {mor1kx_cpu.dbus_err_i == {1'b0}}
# Inputs from the Debug unit
assume -env {mor1kx_cpu.du_we_i == {1'b0}}
assume -env {mor1kx_cpu.du_stb_i == {1'b0}}
assume -env {mor1kx_cpu.du_stall_i == {1'b0}}
assume -env {mor1kx_cpu.du_dat_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.du_addr_i[15:0] == {16'h0000}}
# Internal signals not used in current configuration
assume -env {mor1kx_cpu.spr_bus_dat_pmu_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.spr_bus_dat_pcu_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.spr_bus_dat_mac_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.spr_bus_dat_fpu_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.spr_bus_ack_pmu_i == {1'b0}}
assume -env {mor1kx_cpu.spr_bus_ack_pcu_i == {1'b0}}
assume -env {mor1kx_cpu.spr_bus_ack_mac_i == {1'b0}}
assume -env {mor1kx_cpu.spr_bus_ack_fpu_i == {1'b0}}
assume -env {mor1kx_cpu.snoop_en_i == {1'b0}}
assume -env {mor1kx_cpu.snoop_adr_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.multicore_numcores_i[31:0] == {32'h00000000}}
assume -env {mor1kx_cpu.multicore_coreid_i[31:0] == {32'h00000000}}
```

Listing 5.1: CPU Mode Formal Environment Setup

5.3.2. INSTRUCTIONS INPUT CONFIGURATION

As previously discussed, the test stimuli with higher control over the CPU functionalities is the data supplied by the Instructions Memory **idata_in**; any information applied to such input must respect the definitions from the ISA. Any given CPU includes instructions to control all implemented functionalities; generally, an assembly application requires a sequence of such instructions to execute a functionality correctly. For example, an assembly application that sums up two numbers would need: first, instructions to move the addends to general-purpose registers; second, the addition instruction on such registers; and, finally, the commands to store the result in the memory or return it to a caller function.

In the context of the STL generation, we cannot expect the formal engine to deploy a logical sequence of instructions as exemplified above; instead, it will randomly apply values to the input fields until it finds a combination of values that confirm a property. Therefore, we must include constraints to the formal environment to enable verification using only test stimulus that will result in a concise counter-example. Such conditions must consist of a list of instructions that allow activation of the different CPU blocks without disturbing the software execution flow and boundaries for each instruction's field. For instance, if the counter-example of a formal property includes a *Jump* instruction to a random location, it would not be possible to translate it to a software function; the execution of such operation would result in an exception. Furthermore, we need to preserve the values of registers used to save the original context of an application. For example, in the OpenRISC architecture, we cannot write to general-purpose registers "r1" and "r16" without disturbing software execution; the reason is that the first is a read-only register, and the second holds the return address of caller functions.

For the AutoSoC test case, the analysis for defining the constraints for the formal environment was based on manual verification of each instruction in a simulation environment. The AutoSoC development environment contemplates simulation and software development tools; we have developed a software library function in assembly based on these resources. The initial version of the procedure consisted of an arbitrary execution of assembly commands; first, we integrated it with the AutoSoC Cruise Control application to verify if we could restore the context of the original application after the procedure execution. At this stage, we could identify the general-purpose registers that should be preserved to keep consistency with the main application. Next, we expanded the number of integrated assembly instructions until all were identified by the risk of disturbing the software execution. The instructions that maintain the execution flow are then selected as the base for the STL.

After identifying the commands that will be part of the STL, we need to understand how to configure them in the formal environment. The instructions architecture for the OpenRISC defines several instruction groups, each with distinct bitwise map formats and fields sizes. Furthermore, the configuration of the opcodes must respect the proper format defined in the ISA; in case of mismatches, the formal engine will apply invalid test stimuli to the CPU invalidating the results. Table 5.2 describes the different instruction groups; for each group, it also highlights the definition of the bitwise map and an example assembly command.

Considering the formal environment constraints and the instruction groups illus-

Table 5.2: OpenRISC ISA Instructions Format

Instructions Group	Instruction Format				Assembly Example
1	Opcode [31-26]	D [25-21]	A [20-16]	Value [15-0]	l.addi rD, rA, Value
2	Opcode [31-26] [9-8] [3-0]	D [25-21]	A [20-16]	B [15-11]	l.and rD, rA, rB
3	Opcode [31-26] [9-6] [3-0]	D [25-21]	A [20-16]	B [15-11]	l.sll rD, rA, rB
4	Opcode [31-26] [9-6] [3-0]	D [25-21]	A [20-16]		l.extbs rD, rA
5	Opcode [31-21]	A [20-16]	B [15-11]		l.sfeq rA, rB
6	Opcode [31-21]	A [20-16]	Value [15-0]		l.sfeqi rA, Value
7	Opcode [31-26] [16]	D [25-21]	Value [15-0]		l.movhi rD, Value

trated in Table 5.2, we must construct a logical equation enabling the expected values for the **idata_in** input in the format of an *assume* statement. Such an equation must detail the valid combination of values for each input port bit. The Listening 5.2 demonstrate an example *assume* statement for the **idata_in** input port, including Opcodes from groups 1, 3 and 5 and write restrictions for the general-purpose registers "r1" and "r16".

```
# Set valid opcodes for Formal Analysis
# and add specific Register restrictions per opcode

# Group1 [31:26]:           l.addi, l.andi
# Group3 [31:26][9:6][3:0]: l.sll, l.sra
# Group5 [31:21]:           l.sfeq, l.sfeqi

# OpCodes machine code - For all Opcodes the first register cannot be
# r1 or r16, as both can cause exceptions
assume -env {
# Group1
    ( idata_in[31:26] == {6'b100111} and idata_in[25:21] != {5'b00000}
                                     and idata_in[25:21] != {5'b10000} ) or
    ( idata_in[31:26] == {6'b101001} and idata_in[25:21] != {5'b00000}
                                     and idata_in[25:21] != {5'b10000} ) or
# Group3
    ( idata_in[31:26] == {6'b111000} and idata_in[9:6] == {4'b0000} and
      idata_in[3:0] == {4'b1000} and idata_in[25:21] != {5'b00000} and
      idata_in[25:21] != {5'b10000} ) or

    ( idata_in[31:26] == {6'b111000} and idata_in[9:6] == {4'b0010} and
      idata_in[3:0] == {4'b1000} and idata_in[25:21] != {5'b00000} and
      idata_in[25:21] != {5'b10000} ) or
# Group5
    ( idata_in[31:21] == {11'b11100100000} and idata_in[20:16] != {5'b00000}
                                     and idata_in[20:16] != {5'b10000} ) or
    ( idata_in[31:21] == {11'b10111100000} and idata_in[20:16] != {5'b00000}
                                     and idata_in[20:16] != {5'b10000} )

}
```

Listing 5.2: AutoSoC Opcodes Setup

The final *assume* statement for defining the proper instructions for the AutoSoC analysis included 43 opcodes, together with their restrictions for each field. We have generated several STL versions with smaller subsets of commands to get to this stage. After generating each STL, we have to verify the proper execution of the software flow by integrating it in the Cruise Control application and analyzing the behavior of the CPU in the simulation. The process for generating the STLs will be described in the following sections.

5.3.3. STROBES

The final setup to initialize the formal verification flow is the configuration of the strobes. As previously discussed, the strobes are design components configured as observation points by the user; they indicate to the tool the elements to be monitored to demonstrate fault propagation. In the case of formal analysis, the engine will generate the properties to show fault propagation to the configured strobes; in the context of the STL, the engine will try to identify a sequence of pre-configured opcodes to propagate faults to the observation points.

The common practice for setting up the strobes is to identify all functional outputs of the target design; these are the ports where a fault could propagate to the system and violate safety goals. In this case, the counter-example of the proven properties would indicate test stimuli to reproduce the effects in such outputs. As described in 1.3.1, this class of faults is classified as Dangerous, which can disturb the safety-critical functionalities. However, only propagating faulty elements to the output is insufficient to claim they will become detected; this would require an additional mechanism to verify the design outputs. Ideally, an STL does not require other components to detect faults; the idea is to have a standalone software module able to identify the presence of faults and indicate it to the system. For that reason, we propose a different configuration for the strobes. Figure 5.6 illustrates two strobes configurations on the AutoSoC CPU; in 'a),' the formal setup includes the configuration of the strobes on all functional outputs of the CPU; in 'b),' a single strobe is configured in the general-purpose register "r31".

By modifying the location of the strobes, we are indicating to the tool that we want to identify test stimuli that propagates the effect of faults to the register "r31". Therefore, for each fault analyzed by the formal engine, the counter-example will consist of a sequence of instructions that propagate the outcome to the target register. In other words, after executing such a sequence of commands, the value held by "r31" will differ from the value in the fault-free simulation; this difference indicates the existence of a fault. Furthermore, we can use the content of the register "r31" to calculate a signature; first, we must define the signature during the fault-free simulation; next, during each execution of the STL, we can recalculate the signature. The STL has detected a fault in case of mismatches between the current and the fault-free signatures.

5.3.4. COUNTER-EXAMPLE AND STL GENERATION

Finally, with all required configurations of the environment, we can dispatch the functional safety verification of the formal tool. At the end of the analysis, the engine will classify the fault space elements as Unknown, Safe, or Detected, considering the constraints previously configured. In the cases where the formal engine couldn't finalize the

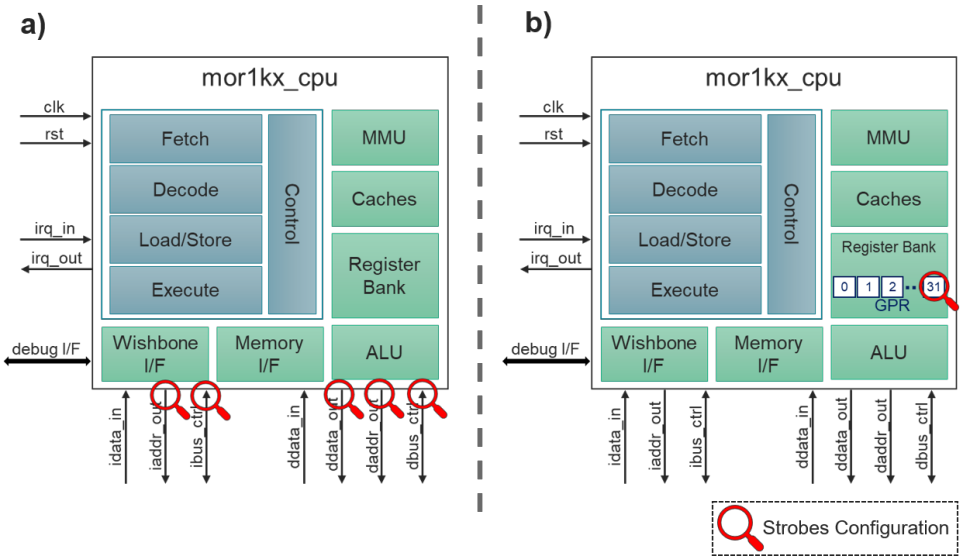

Figure 5.6: a) Strobes configured on CPU outputs b) Strobe configured on GPR r31

analysis due to timing constraints, the faults will be Unknown. If no valid combination of inputs can propagate the fault effect to a strobe, the fault is classified as Safe; it is crucial to note that these faults are Safe only considering the input restrictions; the classification may differ in a constraints-free analysis. Finally, Detected faults are when the tool identifies a valid sequence of instructions propagating it to the strobes. We can extract a counter-example for generating the STL for each Detected fault.

The counter-examples will describe the machine code applied to the **idata_in** input for propagating a given fault to the strobe "r31". As illustrated in Table 5.1, the machine code is a 32-bits numeral representing an assembly command. As we intend to deploy the counter-example result as a software library, we need to translate the machine code to the corresponding assembly instruction. Therefore, we developed a script that inputs a sequence of 32-bits hexadecimal arrays and outputs the related assembly instructions. For example, considering a single Detected fault in the context of the AutoSoC analysis, the item below demonstrates the counter-example extracted from the formal tool.

- 32'hbc480000 32'h9fe007ff 32'he3fffffb 32'hbc420000 32'hbc420000 32'h9fffffff

The sequence of six hexadecimal machine codes represents the instructions for propagating the given fault; we can reproduce this behavior in an STL by converting the machine code into assembly instructions. However, as previously discussed, we intend to detect faults and not propagate them to the outputs. For that reason, we must include the logic for calculating the signature that will indicate the fault propagation; for such, at the end of each sequence of instructions, we manually insert a tag "$SIGN." During the execution of the conversion script, the "$SIGN" will be replaced with a function to handle the signature calculation. The Listing 5.3 demonstrates the assembly code generated

by the conversion script; it contemplates the instructions related to the counter-example and also the call for the signature calculation function.

```
asm("l.sfgtui r8, 0 ");
asm("l.addi r31, r0, 2047");
asm("l.mulu r31, r31, r31");
asm("l.sfgtui r2, 0 ");
asm("l.sfgtui r2, 0 ");
asm("l.addi r31, r31, 65535");
stlSignatureCalc();
```

Listing 5.3: Assembly Code for the STL

The example in Listing 5.3 represents the sequence of instructions for detecting a single fault; as the analysis will incorporate thousands of faults, we cannot verify the signature every time; instead, we must compress the "r31" values and prove it on strategic positions. For such, we apply an Exclusive OR (XOR) between the "r31" current and past values. Furthermore, the function will return the value to the main application after a pre-determined sequence of instructions; the user must define the size of such a sequence based on the application time slots for running the STL. Finally, at the end of each sequence, or STL stage, the compressed signature is returned to the main application to be compared with the fault-free value. The Listing 5.4 describes de source code of the signature calculation function. The function loads the past "r31" values from memory, calculates the XOR with the current value on the target register, and store the result back to the memory.

```
// Inline function to calculate the signature of the STL
void inline stlSignatureCalc(void) {
    // Prepare the higher part of the stlSign address in memory
    // stlSign = (unsigned int *)0x0075000;
    asm volatile ("l.movhi r5, 0x10 ");
    // Load current stlSign value to r30
    asm volatile ("l.lwz r30, 0x5000 ( r5 ) ");
    // XOR r31 and r30
    asm volatile ("l.xor r31, r31, r30");
    // Save r31 to stlSign in memory
    asm volatile ("l.sw 0x5000 ( r5 ), r31 ");
}
```

Listing 5.4: STL Signature Function

After defining the STL stages, we must extract the golden signature values from a fault-free simulation. At the end of each stage, the STL function will return the accumulated signature function result; if the value differs from the golden, it indicates the presence of a fault. Additionally, we must ensure that the CPU's internal values are the same for every call of the STL function. For such, at the start of the function, we manually save the context of the main application and zeroize the general-purpose registers; the context is restored before returning the execution to the main application. Finally, when the main application identifies a discrepancy between the current and golden signatures, it activates an external interruption indicating the detection of a fault.

5.4. RESULTS

The formal analysis for generating the STL for the AutoSoC resulted in 58.676 faults classified as Detected. For each Detected fault, we extract a counter-example and trans-

late it into assembly instruction using the conversion script; these represent a total of 89.495 assembly instructions without counting instructions related to the signature calculation. By analyzing the output of the formal analysis, we identify several repeated sequences of instructions; as the verification of each fault is standalone, every sequence of instructions targets a single fault; therefore, there is no advantage in repeating a sequence in the STL. Furthermore, we can optimize the machine code generated by the counter-examples to decrease the execution time of the STL. For such, we remove repeated sequences of instructions from the counter-examples. After optimization, we achieve 2.282 unique sequences of machine codes, resulting in 19.049 assembly instructions for the final STL.

When analyzing the STL automatic generated by the formal verification, we foresee possibilities for further optimizations. For example, we identified sequences of instructions where the last opcode did not modify the value of "r31". In all cases, at least one command modifies the value of "r31", but in some cases, this is not the last instruction of the sequence. The reason for such behavior is that some instructions require additional cycles to finalize their execution, so the discrepancy caused by the fault is only visible in "r31" after some clock cycles. In the meantime, the formal engine will continue to insert stimuli into the design until the fault is detected. These additional instructions may be worthless for the fault propagation and could be removed from the STL. However, as we target the automatic generation of the STL, avoiding the need for manual efforts from specialists, we decide not to optimize the STL further in the scope of this work.

Next, we must define the length of the STL stages and store the fault-free signature. The final STL includes a total of 19.049 assembly instructions; considering the software cycles of the AutoSoC Cruise Control application, we define each STL stage of around 1.000 instructions, resulting in 19 stages. This size was determined based on the OS time frame and forged after several trials within the simulation environment. Furthermore, we deploy the application simulation, including the activation of the STL, to store the golden signatures for each stage. At each spare frame, the application activates an STL stage; at the end of the execution, the STL returns the calculated signature of the given stage; finally, the application can compare the received and golden values and activate an external interruption, signaling the fault detection, in case of discrepancies.

For verifying the efficiency of the STL, we adopt the Cadence® Xcelium™ Fault Simulator (XFS). The configuration of the XFS includes injection of SA0 and SA1 faults at every cell port of the GTL representation of the AutoSoC CPU. Additionally, the strobes configuration of the FI simulation environment comprises all functional outputs of the CPU as Functional Strobes; the **irq_out** and a Watchdog alarm are set up as Checker Strobes to identify fault detection. Considering the digital area, the AutoSoC CPU contains 96.354 faults for simulation. Table 5.3 describes the results of the FI campaign of the AutoSoC CPU using the proposed STL as the workload. The table shows, for each internal block of the CPU, the number of faults classified as **Unobserved Undetected**, not observed in any strobes; **Dangerous Undetected**, observed only in the Functional Strobes; and, **Dangerous Detected**, observed in the Checker and Functional Strobes. It also highlights the percentage of **Dangerous Detected** faults. The classification of the faults via FI simulation is detailed in 1.3.1.

During the FI Simulation, the STL presents coverage of 53,38% by detecting 51.434 of

Table 5.3: AutoSoC FI Campaign Results

AutoSoC CPU Blocks	Total SA0/1 Faults	Unobserved Undetected	Dangerous Undetected	Dangerous Detected	Dangerous Detected (%)
top	2236	1773	117	346	15,47%
fetch_cappuccino	11460	4977	1041	5442	47,49%
decode	1612	268	41	1303	80,83%
decode_execute_cappuccino	6630	2176	264	4190	63,20%
branch_prediction	20	5	3	12	60,00%
execute_alu	26824	2804	2717	21303	79,42%
lsu_cappuccino	12998	6420	1128	5450	41,93%
wb_mux_cappuccino	1118	148	17	953	85,24%
rf_cappuccino	8398	2219	1421	4758	56,66%
execute_ctrl_cappuccino	3598	904	242	2452	68,15%
ctrl_cappuccino	21460	13818	2417	5225	24,35%
TOTAL	**96354**	**35512**	**9408**	**51434**	**53,38%**

the faults in the CPU. By analyzing the internal blocks of the CPU, we can determine that the proposed approach presents a good coverage for blocks as *decode* and *execute_alu*, while *ctrl_cappuccino* and *lsu_cappuccino* are below the average. One of the reasons for such behavior is that we have restricted the instructions related to control and load-store; as these instructions can disrupt the execution flow of the CPU, it is problematic to deploy them with random parameters generated by the formal engine. Another aspect to note is the average coverage of 56,66% over the *rf_cappuccino* block; in general, manual written STLs achieve higher coverage over the registers; therefore, this can also be considered a disadvantage of the proposed methodology. Even though the coverage of some blocks could be improved by manually including test instructions, the coverage above 50% of all cell ports is considered satisfactory for a software safety mechanism. It is also crucial to highlight that we are only considering Detected faults, as per ISO 26262 definitions; the propagation of faults to outputs is not sufficient to claim a fault as Detected. Furthermore, the numbers presented in Table 5.3 are raw fault injection results; in the next chapter, we will describe the analysis of Safe faults, decreasing the Total Faults and resulting in a higher percentage of Detected faults.

6

ENHANCING THE SAFETY VERIFICATION OF AUTOMOTIVE SOCS

Compliance with ISO 26262 entails a detailed analysis and evaluation of potential random hardware faults. Due to the complexity of these applications, part of this analysis is manually performed by experts, resulting in an expensive, time-consuming, and error-prone process. Furthermore, such a method complicates the analysis of the fault space, making it hard to accomplish ISO 26262 compliance. This chapter proposes an automated method to identify and classify faults overlooked by the traditional flow. Our approach deploys the code coverage to understand the design operational behavior; this behavior is automatically translated into a formal environment, enabling further fault classification. The process is validated on an AutoSoC Benchmark, improving the Diagnostic Coverage and Single Point Fault Metric. Additionally, we perform the functional safety verification of the design according to ISO 26262, validating all methodologies proposed in this thesis and enabling and ASIL C configuration of the AutoSoC.

Parts of this chapter have been published in the IEEE International Test Conference (ITC), 2021 [72].

THE final step for the conception of hardware components in compliance with ISO 26262 is the validation of the development lifecycle and its products. Safety engineers must show evidence that the implemented mechanisms are adequate to comply with the hardware architecture metrics and that the residual probability of safety goal violation is sufficiently low. As previously discussed, a comprehensive fault space analysis is crucial for achieving ISO 26262. Fault analysis and classification according to safety standards is an arduous task that requires extensive knowledge of the design functionalities. Therefore, there is a high demand for methodologies that can speed up fault classification reducing time to market and verification costs.

The industry standard flow for fault classification analyzes SA0 and SA1 faults at all inputs and outputs of the design gates. The philosophy behind it is to identify the functional behavior chance caused by the fault in the DUT; in cases, it cannot disturb safety-related functionalities, the fault is **Safe**; faults that upset the functionalities are **Dangerous**, or **Detected** if SMs are implemented; when the effect provoked by a fault is unknown they are named **Unknown** or **Undetected**. Figure 6.1 illustrates the process of fault classification and analysis for ISO 26262. The process starts with the definition of the fault space; initially, all faults are classified as **Unknown**. Next, Formal Methods are deployed for the identification of Safe faults. After that, FI Simulation is applied to the remaining faults to evaluate the efficiency of SMs; the FI Simulation classifies the faults as **Detected** when SM covers them and as **Undetected** otherwise.

Figure 6.1 also illustrates an additional flow step that is required when deployment of Formal Methods and FI Simulation is not sufficient to classify the whole fault space. The Expert Judgement is an alternative method to classify the Undetected residual faults. Reducing the number of Undetected faults contributes to improving the DC, hence, to ISO 26262 compliance. Nevertheless, relying upon *manual* analysis by experts is not sustainable; the process is expensive, time-consuming, and prone to errors. Consequently, an automated and reliable methodology that decreases manual efforts while fulfilling ISO 26262 requirements is needed.

This chapter presents a methodology setting steps toward fully automated fault space analysis for ISO 26262. Our focus is on identifying the nature of each fault in the fault space, i.e., faults concerning or not concerning safety-critical outputs. For example, suppose the effect of a fault does not affect safety-related functionalities. In that case, there are no Safety Goal violations; hence the fault can be classified as a Safe fault, increasing compliance to safety standards. Initially, we deploy code coverage techniques to identify design elements that are not operated during functional verification. Then, the code coverage reports are examined by an automated tool that generates formal properties to reproduce the observed behavior. Finally, we configure all the properties in a Formal analysis tool, improving the tool's efficiency. The additional classification causes an immediate increase in the Safety Metrics, enabling compliance with ISO 26262. Finally, the methodology is validated using the AutoSoC automatically generated STL, described in 5, as the test case. Furthermore, we validate all methodologies proposed in this work by performing the functional safety verification of the AutoSoC following ISO 26262 requirements; the process includes the FMEDA, Failure Rate analysis, and Safety Metrics calculation. The deployment of this work methodologies enables an ASIL C configuration of the AutoSoC without hardware redundancy.

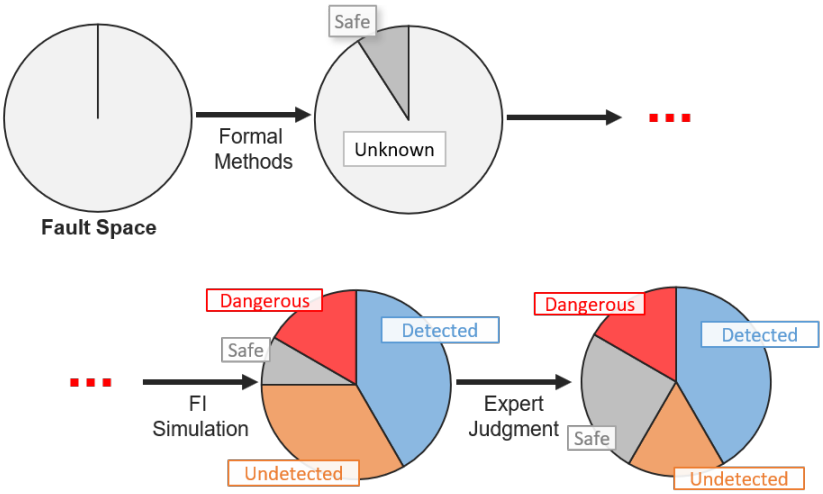

Figure 6.1: Fault Classification Flow

6.1. TESTABLE SAFE FAULTS IDENTIFICATION

The ISO 26262 Hardware Architectural Metrics determines the effectiveness of designs to cope with random hardware failures [34]. The failures addressed by these metrics are limited to elements that can contribute to the violation of safety goals; these define the required mitigation of hazardous events to avoid unreasonable risks caused by malfunctions. During the system development phase, safety goals are decomposed into a Functional Safety Concept that defines the requirements for the hardware architecture. However, the development of a hardware design demands additional components that are not related to the safety concept; these components will decrease the compliance to Hardware Architectural Metrics, even though they may not violate safety goals in case of faults. If that is the case, these components can be identified by their potential to disrupt safety goals, increasing Safe faults classification. Safe faults can be divided into two categories:

- Untestable: there is no combination of test stimuli that induce the fault to affect the functionality of the design. Also know as redundant in the DfT community.
- Testable: faults that can affect the output of the design with a suitable test stimulus. Nevertheless, they cannot affect safety-critical functionalities.

The Testable Safe faults do not cause any deviation of the safety-related operational mode. However, identifying the nature of faults (being safe or not) typically requires the judgment of hardware design experts; this is time-consuming and prone to errors. For that reason, we propose an automated flow to identify Testable Safe faults without relying upon manual analysis. Figure 6.2 illustrates the fault space analysis flow; however, instead of deploying Expert Judgement to identify additional Safe faults, our methodology employs the Automated Flow to identify Testable Safe faults.

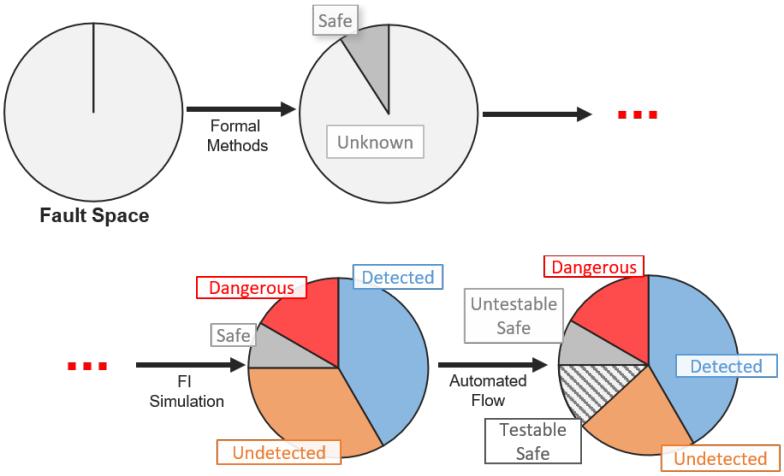

Figure 6.2: Proposed Fault Space Analysis Flow

Figure 6.3 shows our methodology, which explores the strengths of code coverage and Formal Analysis to generate an automated fault analysis flow, decreasing the necessity of manual efforts. Our approach deploys the code coverage to understand the design operational behavior; this behavior is automatically translated into formal properties. By reproducing the design operational behavior in a Formal tool, we decrease the space exploration, allowing the classification of additional faults. Next, the main steps of the method will be explained.

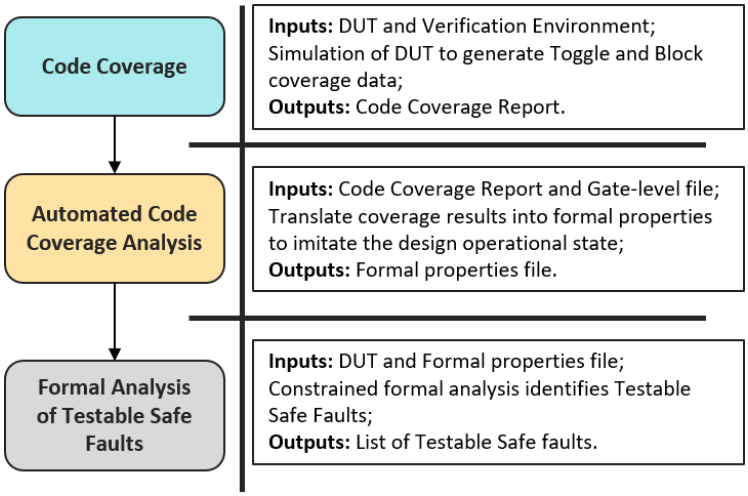

Figure 6.3: Proposed methodology flow

6.1.1. CODE COVERAGE

Code coverage is a method of assessing to what extent test cases exercise a design under test. The method deploys simulation of a target design considering a group of test cases; it is crucial to consider a comprehensive batch of test cases to ensure the code coverage results represent the actual design functionalities. Furthermore, as safety verification is performed after the functional verification, we can assume that the verification environment will be available for the testable safe fault analysis.

The code coverage analysis will verify the variations in all internal signals of the target design, resulting in detailed reports on the behavior of such signals. In the context of this work, we will utilize the block and toggle coverage reports. Block coverage determines whether test scenarios exercise the statements in a block. A block is a series of sequential statements without delays or control flow statements (if, case, wait, while, among others). In other words, a block is a specific state in a state machine. Toggle coverage measures the activity of the signals in the design during the simulation. It provides information on untoggled signals or signals that remain constant during the simulation.

The metrics from the code coverage provide information regarding design parts that may not be safety-related. For instance, by recognizing states that are never activated due to block coverage, we can identify design modes that are not related to safety functionalities. Similarly, untoggled signals can highlight important details of the design, like invalid configurations, not utilized functions, status monitors, among others. The combination of toggle and block coverage provides additional information about specific functionalities. For example, the missing toggle in a control signal may be responsible for never activating a block in a state machine. Also, by bypassing a specific state, another signal may not be toggled. Figure 6.4 illustrates an example of the correlation between the toggle and the block coverage. The block coverage (Figure 6.4-a) shows a never activated block. Since the last "else if" statement is always false, the 'error_irq' is not set to zero. In Figure 6.4-b, the result of toggle coverage shows that the control signal 'read_irq_reg' never toggles, validating the block coverage. Additionally, the coverage confirms that the signal 'error_irq' has one rising toggle but never toggles back to zero.

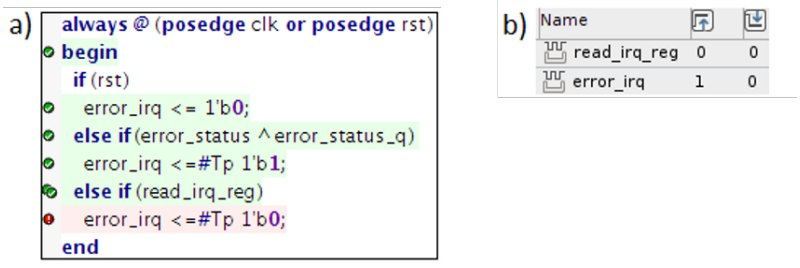

Figure 6.4: a) Block Coverage example - b) Toggle Coverage example

In this example, the code coverage result triggers an investigation that concludes that the interrupt requests (IRQ) error register is never read. In this case, a fault in the IRQ error register can not disturb safety-related functionalities; hence it is a Safe fault. If we can reproduce the described behavior in a formal tool, we can automate the identification of Safe faults as the IRQ error register.

6.1.2. AUTOMATED CODE COVERAGE ANALYSIS

The automation process aims to translate the code coverage behavior into formal properties. For example, by replicating the control signals' logic values from Figure 6.4 in the formal environment, we reduce the space exploration, allowing formal to identify the IRQ error register as a Safe fault. As the formal environment includes operational constraints, the newly identified faults are Testable Safe faults.

The automated code coverage analysis tool will translate design elements from the code coverage report into *assume* statements or *fault-propagation barriers*. *Assume* statements enable constraints configuration for formal analysis. When an expression is assumed, the formal verification tool constrains the design inputs accordingly. The role of the *Assume* construct is useful in the confirmation of the design functional configuration. Also, by configuring the expected behavior of the design, we increase the capacity of Safe faults identification by limiting the test stimuli space. *Fault-propagation barriers* are design elements that can block the propagation of a fault. Faults that propagate only to certain elements may not affect safety-critical functionalities. For example, any fault that can only propagate to an IRQ output port disconnected (open) in the design implementation cannot disturb safety-related functionalities. Consequently, these faults can be Testable Safe faults.

The automated code coverage analysis must also examine the design element types and values. For example, input ports of the design instances are suitable candidates to *assume* statements; output ports, on the other hand, are better candidates for *fault-propagation barriers*. The tool automatically retrieves the signal types and values by analyzing the coverage report and source code. An illustrative description of the automated code coverage analysis tool implementation is described in Algorithm 1. The Algorithm describes the logic executed for analysis of the toggle and block coverages; the definition of the *assume* statements, and *fault-propagation barriers* depend upon the results of the report and the declaration of relevant signals as extracted from the source code. During the loop that describes the analysis of the block coverage report, we have simplified that analysis of the block control with a single command, **get(control_signals)**. However, it is essential to highlight that such an analysis requires an interpretation of the logic that defines the block flow. For example, in the case of a statement checking the result of a logic "AND" between two signals; we need to identify the value of the signals so that we can define the reason why the "true part of" the statement is never true, and replicate this behavior as an *assume* statement.

Next, the automated code coverage analysis will output a text file containing all formal properties generated for the design. Listing 6.1 illustrates an example of such output. The output includes the formal properties (*assume* statements and *fault-propagation barriers*) and supplementary information that lead to the creation of the properties. In the example, as the code coverage shows no activity in the Debug Unit (du), its inputs are always zero, and we can ignore faults that propagate to it; this behavior is described as an *assume* statement declaring that debug input is always zero, and a *fault-propagation barriers* on the debug data output.

The only required manual step is revising the formal properties generated by the automated code coverage analysis tool. The coverage result is not enough to distinguish the potential Testable Safe faults; for such, it would need to include information related

Algorithm 1 Automated Code Coverage Analysis Tool

Input: toggle_coverageReport, block_coverageReport, netlist

```
read(coverageReports)                                   ▷ Read reports
read(netlist)                                           ▷ Read design information

while (getLine(toggle_coverageReport) ≠ −1) do          ▷ Toggle Coverage Analysis
    if (signal is input) then                           ▷ assume statement
        if (signal toggle is always 0) then
            set assume signal == 0
        end if
        if (signal toggle is always 1) then
            set assume signal == 1
        end if
    else if (signal is output) then                     ▷ fault barrier
        if (signal toggle is always 0) then
            set fault_barrier signal
        end if
    end if
end while

while (getLine(block_coverageReport) ≠ −1) do           ▷ Block Coverage Analysis
    get(control_signals)                                ▷ Get Block control signals
    if (blockCoverage = "true part of") then            ▷ True part is never activated
        set assume signal != (block control TRUE)
    else if (blockCoverage = "false part of") then      ▷ False part is never activated
        set assume signal != (block control FALSE)
    else if (blockCoverage = "a case item of") then     ▷ Case never activated
        set assume signal != (block control CASE value)
    end if
end while
```

to the functional requirements of the design. To ensure the formal property does not conflict with the expected behavior, engineers must review the output of the automated code coverage analysis tool. For that reason, as illustrated in Listing 6.1, the tool output includes supplementary information to support the review process. Each formal property generated by the tool includes a comment section with the rationale for defining such property; it includes comments describing the instances information, declaration and connection of the signals in the source code, and the code coverage results that lead the investigation. Each formal property must represent the reality of the design. An over-constrained formal environment would cause false positives, invalidating the results.

```
# ======================================
# Toggle Coverage Analysis Result
#
# Instance Name: mor1kx_cpu.mor1kx_ctrl
# Source Code: mor1kx_ctrl.v
#
# ======================================
# List of Signals with no Rise and no Fall
# (always 0)
# ======================================

# Assume Proposal
assume -env {mor1kx_ctrl.du_dat_i[31] == 1'b0}

# ======================================
# List of Outputs with no Toggle Activity
# for Fault Barriers Analysis
# ======================================
# Instance Name: mor1kx_cpu.mor1kx_ctrl
# Module Source Code: mor1kx_ctrl.v
# Top Source Code: mor1kx_cpu.v

# ======================================
# Match of du_dat_o on Top Source Code
# Source: mor1kx_cpu.v
#   148     output du_dat_o,
#  1438       .du_dat_o     (du_dat_o),

# ======================================
# Match of du_dat_o on Module Source Code
# Source: mor1kx_ctrl.v
#   198     output du_dat_o,
#  1282      assign du_dat_o = du_read_dat;
#  1449        assign du_dat_o = 0;
# ======================================

# Barrier Proposal
check_fsv -barrier -add {mor1kx_ctrl.du_dat_o}
```

Listing 6.1: CPU Mode Formal Environment Setup

6.1.3. FORMAL ANALYSIS OF TESTABLE SAFE FAULTS

The Testable Safe faults identification is based on the traditional formal verification flow to identify Safe faults. The difference is that the formal environment will incorporate the results of the automated code coverage analysis tool. By constraining the environment, we enable the tool to evaluate the design in a well-specified configuration; the constraints decrease the number of possible test stimuli combinations, increasing the potential for identifying Safe faults. Furthermore, we need to differentiate between Testable and Untestable Safe faults; for such, we deploy the formal verification flow twice. First, without the automatically generated constraints, the traditional flow identifies Untestable faults. Next, the verification is repeated, including the constraints. Additional Safe faults, identified in the second run, will be classified as Testable Safe, as they are only Safe when considering the functional constraints included in the environment.

To enable the identification of the Testable Safe faults, we must source the output of the automated code coverage analysis tool into the formal environment. The set-up file must include all *assume* statements and *fault-propagation barriers*; also, it must be reviewed to ensure the formal properties do not contradict functional requirements,

avoiding over-constraining the design. Finally, after properly configuring the formal environment, we repeat the formal verification flow, resulting in additional Safe faults; such faults can be classified as Testable Safe.

6.2. RESULTS

6.2.1. TEST CASE

The validation of the proposed methodology targets the AutoSoC STL test case previously described in 3 and 5. Such configuration of the benchmark includes ECC protection on the internal memories and the automatically generated STL as protection to the digital portion of the CPU. As conclusive evidence of the proposed methodology, we must follow ISO 26262 functional safety verification guidelines:

1. Analyze the contribution of the ECC and STL SMs; for the STL, we must recalculate the Diagnostic Coverage (DC), including the assistance of the Testable Safe faults.

2. Assess the CPU failure modes and their contribution to the failure rate; we must consider the safeness and detection rates to compute the residual failure rates for each failure mode.

3. After completing the Failure Modes Effects Diagnostic Analysis (FMEDA), we can calculate the safety metrics and verify compliance with ASIL.

The internal memories occupy the highest area of the AutoSoC physical device, representing 91,3% of the fault space. Based on ISO 26262 standard recommendations, we can assume the ECC provides a Diagnostic Coverage of 99% for random hardware faults on internal memories, representing a satisfying coverage for the overall CPU. The AutoSoC CPU includes 633.344 fault targets in the internal memory cells. Considering the DC of 99%, we can conclude that the ECC covers 627.011 faults, while 6.333 faults are still Undetected.

For verifying the efficiency of the STL, we must revisit the FI simulation results described in Chapter 5. Considering only the digital area, the AutoSoC CPU contains 96.354 fault targets for Simulation. During the FI Simulation, the STL presents coverage of 53,38% by detecting 51.434 of the faults. The analysis also shows that 35.512 faults are **Unobserved Undetected**; from these, we can identify Safe faults. The traditional formal analysis flow identifies 8.020 Untestable Safe faults; these can be decreased from the Total number of faults for DC calculation, resulting in a detection rate of 58,23%.

In a preliminary analysis of the Safety Mechanisms listed above, we can conclude that the memory area has sufficient coverage with the DC provided by ECC. However, the digital area of the CPU may still require further coverage for achieving the required safety metrics. Furthermore, even though the deployed STL achieves significant DC, over 27.000 faults are still **Unobserved Undetected**. These faults must be classified to allow compliance with the requirements of ISO 26262. The following section shows how the proposed methodology impacts the Diagnostic Coverage of the STL by classifying **Unobserved Undetected** faults as Testable Safe.

6.2.2. CLASSIFICATION OF TESTABLE SAFE FAULTS

The first step for the identification of the Testable Safe faults is the code coverage analysis. The AutoSoC code coverage analysis consisted of the simulation of the 100 workloads available in the benchmark. The workloads cover a variety of applications, producing a representative baseline of the CPU functionalities. Next, the automated code coverage analysis tool is deployed on the code coverage report to generate the next step's formal properties. We investigated the properties file to avoid over-constraining the formal analysis. The revision process included inspection of the RTL code and monitoring of the signals during the simulation. Also, some RTL internal signals needed to be traced to wires in the Gate level representation of the hardware. The final formal properties file consisted of 3.884 *assume* statements and 63 *fault-propagation barriers*.

Our work applies Cadence® Integrated Metrics Center (IMC) for code coverage and Cadence® JasperGold (JG) Formal Verification Platform Functional Safety Verification (FSV) for Formal Analysis. The identification of Safe faults consisted of two steps. First, we deploy JG FSV formal analysis for the identification of Untestable Safe faults. Next, we load the formal properties file into the Formal Analysis tool and repeat step one. The additional Safe faults identified in step two will be listed as Testable Safe. The computational time required for each Formal campaign was a couple of days. As many properties are never proven, the total execution time depends on the timeout configured for each formal property.

Table 6.1 demonstrates the results for each analysis step. First, formal verification identifies Untestable Safe faults; next, we repeat the flow, including the functional constraints for determining Testable Safe faults; finally, we deploy FI simulation to classify the remaining faults.

Table 6.1: AutoSoC Fault Space Classification Results

AutoSoC CPU Blocks	Total Faults	Untestable Safe	Testable Safe	Total for FI	Unobserved Undetected	Dangerous Undetected	Dangerous Detected	Dangerous Detected (%)
top	2236	807	907	522	59	117	346	66,28%
fetch_cappu	11460	1030	0	10430	3947	1041	5442	52,18%
decode	1612	5	0	1607	263	41	1303	81,08%
dcde_exec_cap	6630	632	42	5956	1502	264	4190	70,35%
branch_predict	20	0	0	20	5	3	12	60,00%
execute_alu	26824	31	667	26126	2106	2717	21303	81,54%
lsu_cappu	12998	708	66	12224	5646	1128	5450	44,58%
wb_mux_cap	1118	44	64	1010	40	17	953	94,36%
rf_cappuccino	8398	2	0	8396	2217	1421	4758	56,67%
exec_ctrl_cap	3598	208	67	3323	629	242	2452	73,79%
ctrl_cappu	21460	4553	1034	15873	8231	2417	5225	32,92%
TOTAL	**96354**	**8020**	**2851**	**85483**	**24641**	**9408**	**51434**	**60,17%**

Figure 6.5 illustrates the results highlighting the contribution of the various analysis steps. The graph illustrates the faults classification contribution achieved during Fault Injection, Untestable Safe (UF), and Testable Safe (TS) analysis. The process is incremental, always focusing on faults that were previously Undetected. Also, Figure 6.5 displays the calculated Diagnostic Coverage at each step. The proposed methodology can identify 2.851 additional Safe faults, resulting in a 1,94% increase in the Diagnostic Coverage. As previously explained, we apply Formal methods to decrease the number of Undetected faults. As increasing the number of Safe faults decreases the denominator in the

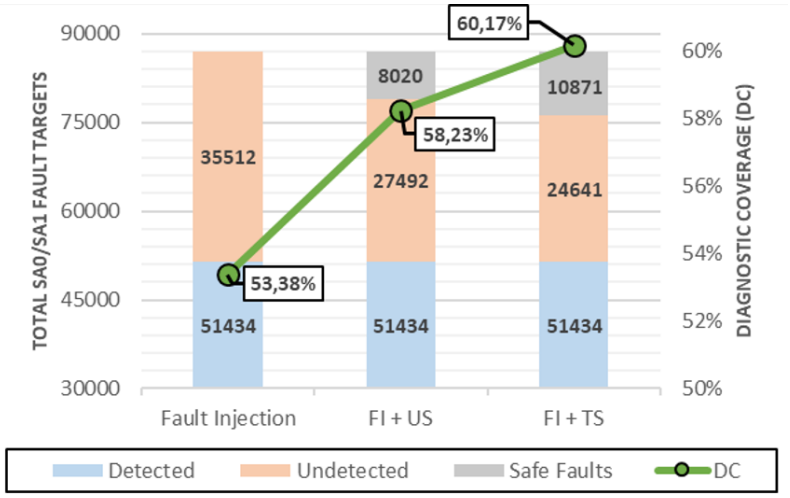

Figure 6.5: STL Diagnostic Coverage Analysis

DC equation (1.5), our methodology directly influences the Diagnostic Coverage.

6.2.3. FUNCTIONAL SAFETY VERIFICATION

The calculation of the Diagnostic Coverage is an indication of the design safety based on the efficiency of each Safety Mechanism. However, to assure compliance with ISO 26262 requirements, a comprehensive Functional Safety Analysis must be performed. The analysis intends to confirm that the probability of failures in a safety-relevant system is reduced to acceptable levels. The primary methodology for the Functional Safety Analysis of hardware devices is the Failure Modes Effects and Diagnostic Analysis (FMEDA). The FMEDA correlates IC components (Gates, Flops, and Memory cells) to Failure Modes (FM). Also, considering the base Failure In Time (FIT) rate of individual IC components, the Diagnostic Coverage of Safety Mechanisms, and the Safe faults, we can determine the Residual FIT contribution of each FM. The sum of the FM FIT contributions is essential to calculate the Safety Metrics.

The FMEDA starts with the definition of the FMs and the mapping of design components. For the AutoSoC, we considered ten subparts, one for each sub-block of the CPU. Each subpart includes the FMs for Data Corruption, Deadlock, Modified Execution, Exceptions, and Performance. After, we must connect each FM to the appropriate design components; the calculation of the FMs FIT is related to the design components mapped to it. For that reason, each FM must include the total area, number of gates, flops, and memory bits. The AutoSoC FMEDA comprises 75 Failure Modes mapped to 28.956 Gates, 1.983 Flops, and 316.672 Memory cells. Table 6.2 demonstrates an example failure mode mapped to the *ctrl_cappuccino* design block; it also includes the design block information extracted from the netlist.

The Safety Metrics calculation consists of the contribution of each FM to the FIT Rate (λ). To calculate λ, we need to define the base λ for a given tape-out technology. Gen-

Table 6.2: FMEDA: Failure Mode and Area

ID	Part	SubPart	Failure Mode	Technology	Area	#Gates	#Flops	#Bits
FM_1	cpu_top	ctrl_cappuccino	Data Corruption	Digital	1348,2	1314	107	0

erally, the definition of the base λ is based on the historical data of a given technology; another option is the calculation based on reliability standards. For example, the IEC 62380 Electronic Reliability Prediction Standard defines a base λ for several technologies and components. Therefore, following IEC 62380 guidelines, we can determine the technology λ of the Digital components (based on the equivalent area of a NAND2 gate) and of the Memory components (based on the area of a memory cell). Then, the λ of each FM is calculated by multiplying the mapped design area by the base λ for each technology. Finally, following the example of Table 6.2, we can multiply the **Area** by the base λ for the chosen technology, resulting in the Raw λ of the FM.

The calculation of FM λ must also assess the contribution of Safe faults and the percentage of faults protected by SMs. The fault space analysis determined the components of the CPU blocks that could not disturb safety-related functionalities; the FIT related to these Safe components should be decreased from the FM λ. For example, we have previously determined that 26% of the faults in the *ctrl_cappuccino* are Safe; therefore, we can conclude that the 26% of the Raw λ cannot disrupt safety-related functionalities. Likewise, we have determined that the STL achieves a detection rate of around 33% for the *ctrl_cappuccino* block; the faults that SM detects also cannot disrupt safety-related functionalities and must be accounted for accordingly. Finally, the FM Residual λ is calculated as the Raw λ minus the λ contribution of components identified as Safe and Detected. Table 6.3 illustrates the components for the computation of the λ, as a continuation of the **FM_1** from Table 6.2.

Table 6.3: FMEDA: FIT Rate Parameters

ID	Raw FIT	Safe Faults(%)	Safe FIT	Non-Safe FIT	SM ID	Diagnostic Coverage(%)	Residual FIT
FM_1	7,536e-02	26,0%	1,962e-02	5,574e-02	STL	32,92%	3,739e-02

We can also decompose the FIT according to ISO 26262 definition of fault classes; every portion of the λ provokes a behavior, the different classifications represent that. The λSPF represents Single-Point faults that SMs do not cover; these are hardware components without correlated SM. The Residual (λR) describes elements with SM coverage but classified as Undetected during the fault space analysis. On the other hand, the Detected faults, which could only violate a safety goal combined with a second fault, are called Multi-Point faults (λMPF). λS represents the contribution of Safe faults. The sum of the fault classes is equal to the total λ, as defined in the equation 6.1.

$$\lambda = \lambda SPF + \lambda R + \lambda MPF + \lambda S \tag{6.1}$$

The classification of the λ classes is necessary for determining ASIL compliance. The ASIL requirements are expressed as target values in the form of metrics. These metrics

are calculated based on the fault classes' contribution to the total λ [73]. The metrics defined by ISO 26262 are the Single-Point Fault Metric (SPFM), the Latent Fault Metric (LFM), and the Probabilistic Metric for Random Hardware Faults (PMHF).

The Single-Point Fault Metric (SPFM) considers λSPF and λR potentials to violate safety goals. In other words, it defines the probability of violation due to faults in design elements without SM coverage or Undetected by SM. The SPFM is calculated according to the equation 6.2.

$$SPFM = 1 - \frac{\sum(\lambda SPF + \lambda R)}{\sum \lambda} \tag{6.2}$$

The Latent Fault Metric (LFM) considers faults that cannot directly violate a safety goal but could be a risk in the presence of a second fault, i.e., Latent Multi-Point faults (λMPF,L). The λMPF,L are faults that can only violate a safety goal in connection with a second fault (i.e., faults in a Safety Mechanism). For this calculation, we must subtract the λSPF and λR from the overall λ. The equation 6.3 defines the LFM calculation.

$$LFM = 1 - \frac{\sum(\lambda MPF,L)}{\sum(\lambda - \lambda SPF - \lambda R)} \tag{6.3}$$

The PMHF evaluation provides evidence that the cumulative safety target violating failure rate of all hardware elements is sufficiently low. Unlike the other metrics, ISO 26262 provides multiple methods to perform the PMHF analysis and does not define an equation. For that reason, we refer to IEC 61508 Safety Standard for the PMHF calculation as in 6.4.

$$PMHF = \sum \lambda SPF + \sum \lambda R + \sum \lambda MPF,L \tag{6.4}$$

Figure 6.6 details the result of the Functional Safety Analysis of the AutoSoC. The left axis presents the classification of FIT's different λ classes, while the right axis demonstrates the SPFM final result. The analysis considered four safety configurations:

1. ECC: Includes the DC only of the ECC applied to Failure Modes connected to the internal memories;
2. ECC+STL: Incorporates the STL SM in the prior version, adding DC to the digital area of the CPU;
3. ECC+STL Untestable Safe: Comprises the previous versions, including the Untestable Safe percentage for each FM;
4. ECC+STL Testable Safe: Includes all features of the versions above and increases the Safe percentage according to the Testable Safe results.

When comparing the results achieved by versions ECC and ECC+STL, we can observe the impact of the automatically generated STL in the safety metrics. Even though the ECC enables a high coverage on a significant area of the design, the lack of SMs in the digital area limits the potential for achieving higher ASILs. Also, we can observe that the ECC is the only version with λSPF different from zero; as the other versions include SMs in all FM, the λSPF becomes zero. Furthermore, all versions result in the same total λ, resulting from the sum of the other λ classes.

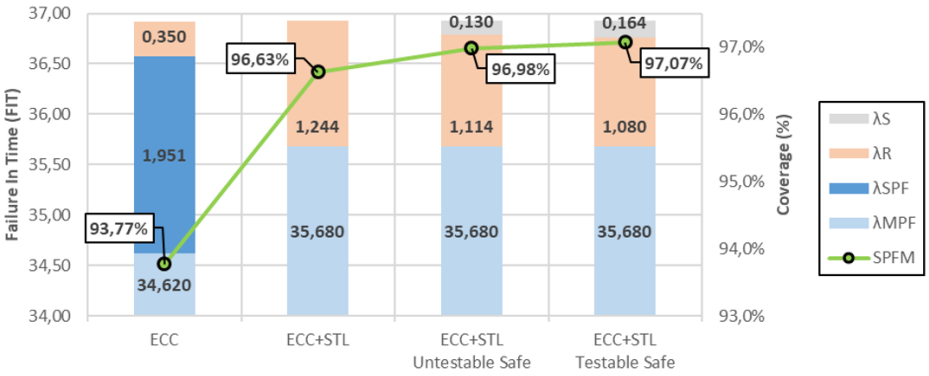

Figure 6.6: Safety Metrics Analysis

The graph also illustrates how the increase in the λS directly impacts the SPFM. The increment in λS causes a decrease in λR and, therefore, enables an SPFM coverage of 97,07%. Furthermore, as the analysis has not contemplated hardware components responsible for Safety Mechanisms, the λMPF,L is zero; consequently, the LFM result is 100% for all FMEDAs. Finally, as described above, the λSPF and λMPF,L are zero for all Failure Modes; hence, we can determine from the equation 6.4 that PMHF is equal to the total λR.

Table 6.4 describes the minimum Safety Metrics requirements for each ASIL according to ISO 26262; also, it shows the calculated metrics for the AutoSoC FMEDA when applying the proposed methodologies. The additional coverage provided by the Testable Safe faults enabled an SPFM of 97,07%, achieving the minimum requirement for an ASIL C design. It is crucial noting that, as seen in Figure 6.6, our results allowed an immediate increase in the ASIL of the AutoSoC.

Table 6.4: Safety Metrics Requirements and AutoSoC results

Safety Metrics	ASIL D	ASIL C	ASIL B	AutoSoC
SPFM	≥ 99%	≥ 97%	≥ 90%	**97,07%**
LFM	≥ 90%	≥ 80%	≥ 60%	**100%**
PMHF	10 FIT	100 FIT	100 FIT	**1,08 FIT**

Even with the increased fault classification, there is still a considerable contribution from λR (Undetected faults), which decreases the coverage restricting ASIL D compliance. The classification of the Undetected faults could be achieved by improving the STL coverage, including additional Safety Mechanisms, or adjusting the design analysis to increase the number of Safe faults. In complex designs, it is challenging to achieve 99% of SPFM without hardware modifications. The traditional industrial-grade Automotive CPUs deploy redundancy schemes, as the Dual-Core LockStep (DCLS), to achieve ASIL D. Even though we cannot achieve ASIL D, our work significantly contributes to ASIL

compliance. This contribution is an essential factor for improving the safety of the CPU without the need for redundancy schemes.

The proposed methodology appears as a promising alternative for the classification of Undetected faults. We define a systematic approach that allows the identification of Safe faults based on two well-established techniques: code coverage and formal verification. Identifying these faults usually relies on reliability experts and requires deep knowledge of the system functionalities; such a process is strenuous and prone to errors. Our automated approach classifies 2.851 additional faults, improving the STL DC by 1,94%. The additional coverage enables a final SPFM of 97%, enabling the AutoSoC to achieve the requirements for an ASIL C hardware component as-is, i.e., without design modifications.

7

CONCLUSION

This dissertation addresses the role of the EDA industry in supporting the safety aspects of automotive electronic systems. Nowadays, the development of automotive semiconductors must respect safety standards like ISO 26262. Such standards define requirements to assure functional safety verification; the development considered measures to confirm that the probability of failures is acceptable. As one of the key players in the semiconductor industry, the EDA industry has a crucial role in enabling functional safety verification. Furthermore, as the safety lifecycle defined by ISO 26262 comprehend various phases of the development and verification of automotive semiconductors, we propose methodologies for different development stages. This chapter presents an overview of the tools and methodologies proposed in this dissertation; it highlights contributions to the safety lifecycle and details each methodology individually.

THE prospect of fully autonomous vehicles promises to revolutionize mobility concepts in the coming years. As a result, the automotive industry is investing heavily; the semiconductors' revenue will potentially reach 50 billion dollars shortly. This favorable scenario generates momentum for investments aiming to advance the technologies embedded in a vehicle. As one of the key players when considering semiconductors, the Electronic Design Automation (EDA) industry has a crucial role. Nowadays, EDA companies provide several products to assist the traditional semiconductors development flow and tailored solutions to address the needs of specific industries. Nonetheless, the concept of autonomous vehicle applications implies new challenges, as a life-threatening situation caused by a malfunction in electronic systems is unacceptable.

This dissertation addresses the role of the EDA industry in supporting the safety aspects of automotive electronic systems. We propose methodologies to deploy the traditional EDA technologies into functional safety verification, improve compliance to ISO 26262, and ensure the safety integrity levels of automotive devices. Furthermore, in this Ph.D. project, we address the challenges at different stages of the safety lifecycle proposing methodologies to support the development and verification phases. Figure 7.1 demonstrates how the main contributions of this dissertation support the safety lifecycle as defined by ISO 26262.

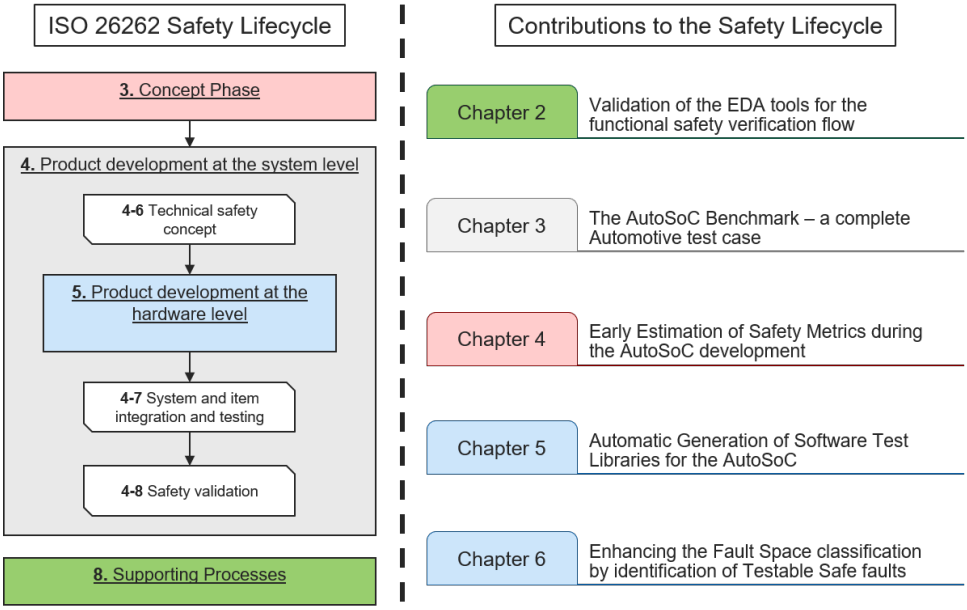

Figure 7.1: Contributions of the thesis to the ISO 26262 safety lifecycle

7.1. FINDINGS OVERVIEW

Next, we will emphasize the concepts and findings of each proposed methodology to the safety lifecycle. Furthermore, as the presented methods are organized in the different chapters of this dissertation, we will present the following conclusions preserving the same organization.

CHAPTER 2: FUNCTIONAL SAFETY VERIFICATION METHODS AND VALIDATION

This chapter introduces the functional safety standard demands for validating the software tools used during the safety lifecycle; such requirements aim to ensure that the level of confidence in the usage of these engines is adequate. In other words, they cannot mask or fail to identify failures in the design. We propose a methodology to improve the Tools Confidence Level (TCL) by detecting malfunctions in the tools used for fault analysis; for automating the execution of the methodology, we conceive the Fault Checker application. The application deploys the analysis of test cases with three different fault classification technologies: Automatic Test Pattern Generators (ATPG), Formal methods, and FI simulators; by comparing the fault annotation from the various technologies, we can identify possible malfunctions. Furthermore, the Fault Checker enables the use of test environments generated by ATPG for the FI simulation, allowing similar conditions for the fault analysis in the different technologies; and avoiding efforts with the development of test benches. Finally, the application generates a fault report to identify potential software tools malfunctions. The inclusion of redundancy as a method to detect malfunctions in tools is a suggested method for achieving ISO 26262 Tool Confidence [33]. The validation of the development environment and software tools is a prerequisite for compliance with the safety life cycle requirements.

CHAPTER 3: SAFETY BENCHMARKS FOR AUTOMOTIVE SOCS

This chapter outlines the conception and development of the AutoSoC Benchmark Suite; it describes the current architecture options, including hardware components, software applications, operating systems, and safety mechanisms. Due to the current demands for research targeting autonomous driving solutions, the availability of industrial-grade test cases is crucial. Nonetheless, the limited access to representative designs and industrial methodologies poses a challenge to the research community. Therefore, the AutoSoC intends to provide researchers with an automotive SoC based on commercial solutions, including all essential components, highly customizable and allowing comparability between distinct methodologies and results. Furthermore, we believe that the availability of this benchmark suite will allow researchers to develop new solutions and quantitatively assess their effectiveness, thus contributing to the advancement of the state-of-the-art in the several technologies required by these applications, e.g., safety, security, performance, among others. This chapter also describes the preliminary functional safety assessment of the AutoSoC configurations targeting different ASIL.

CHAPTER 4: EARLY ESTIMATION OF DESIGN SAFETY METRICS

This chapter discusses the challenges of early estimation of fault space classification. At later stages of safety-critical systems development, designers must analyze the behavior of the design under the effect of faults to show conformity with the expected safety

metrics. Failing to achieve these conditions entails additional iterations through critical development and verification phases. Furthermore, by allowing engineers to estimate safety metrics at earlier development stages, we provide a tool for investigating safety architectures, improving the confidence in conceptual decisions, and decreasing the chances of rework. We tackle this issue by allowing the estimation of Detected faults throughout the safety lifecycle. We consider the current hardware abstraction level for each development stage to model faulty behavior; as the development progresses, we enhance the results by computing more accurate hardware descriptions. Furthermore, the technique is based on the characterization of the hardware description to determine how the components contribute to fault propagation. Also, by examining the test stimuli applied during simulation, we can rank Workloads/Testbenches according to their fault detection coverage. Our results demonstrate the accuracy of the technique by providing an estimation of the fault detection rate with an average error of 3%. Moreover, the methodology results in an execution time up to 20X faster when compared with the traditional gate-level Fault Injection campaigns.

CHAPTER 5: ENHANCING ONLINE FAULT DETECTION OF AUTOMOTIVE CPUS

This chapter describes a methodology for improving online fault detection in CPUs without hardware and development overheads. A common approach in safety-critical domains is to deploy redundancy of systems and components, such as the Dual-Core Lock Step (DCLS). However, the overhead costs of redundancy schemes have a substantial impact on automobiles; the consequence on the final price may be prohibitive in such a competitive market. In contrast, safety schemes based on Software-Based Self-Testing (SBST) are a good alternative for improving integrity levels; solutions based on Self-Test Library (STL) - a collection of SBST procedures - are widely deployed in the Automotive industry. However, the manual development of such STLs is very complex, demanding specialized resources and long development cycles. To tackle such a problem, we propose a formal-based technique for generating STLs. First, we constrain the formal environment to use only pre-selected CPU instructions; then, the formal verification will determine a sequence of such instructions to propagate a given fault. Furthermore, we modify the traditional strobes configuration to calculate a signature indicating the presence of faults; such an approach results in a standalone safety mechanism enabling detection of faults. Finally, the technique is validated using the AutoSoC CPU; the automatically generated STL achieves a detection rate of 53% for SA0/1 faults in the CPU digital area.

CHAPTER 6: ENHANCING THE SAFETY VERIFICATION OF AUTOMOTIVE SOCS

This chapter presents an automated methodology that combines code coverage and formal verification techniques for Safe fault identification. The severe demands for tolerance to random faults are a hurdle for ICs targeting ISO 26262 safety-critical applications. As part of this process, fault analysis methods are still driven by experts, requiring manual analysis that is very expensive, time-consuming, and prone to errors. The proposed methodology begins with code coverage analysis for identifying design elements where a fault cannot disturb safety-critical functionalities. Next, those elements are automatically translated into formal rules and configured in a formal analysis environment, enabling the identification of Testable Safe faults. Reducing the contribution of λR (Unde-

tected) faults by identifying additional Safe faults is mandatory for improving the safety compliance of a hardware design. We validate our methodology on the AutoSoC configuration deploying the automatically generated STL. The approach improves the STL Diagnostic Coverage by 6,7% considering the raw FI simulation results and 1,94% when compared with the state-of-the-art approach for identifying Safe faults. The proposed methodology appears as a promising alternative for Undetected faults classification. Our automated approach reduces the constraints of manual expert-based analysis, reducing the verification costs and time to market.

Furthermore, we performed the functional safety verification of AutoSoC STL configuration; the verification includes an FMEDA, Failure Rate analysis, and Safety Metrics calculation, according to ISO 26262 guidelines. Such a process intends to validate all propositions of this dissertation by applying them to an automotive test case, showing actual improvement on the final safety compliance. First, the software tools to be deployed for the functional safety verification are validated according to the Tools Confidence Level (TCL) requirements, as described in 2. Furthermore, we must define the design under development at the system level, as shown in 3; the test case illustrates the challenges and complexity of automotive solutions. After, as the development of the AutoSoC progresses, we deploy the techniques demonstrated in 4 for estimating the safety metrics in various stages of the development lifecycle. Later, in the final stages of the design development, we must find solutions to enable configurations of the AutoSoC without redundancy overhead; for such, we apply the technique presented in 5 to improve online fault detection without additional hardware. Next, the fault space analysis of the AutoSoC STL configuration includes the additional coverage from the identification of Testable Safe faults, as shown in 6. Finally, we validate all the proposed methodologies by performing the complete functional safety verification. The improvements achieved by the additional coverage enable compliance with ASIL C requirements, with 97% coverage of single-point faults. Also, it enables an accurate safety evaluation, allowing compliance to ISO 26262 without hardware redundancy.

BIBLIOGRAPHY

[1] International Standardization Organization, *ISO 26262 Road Vehicles - Function Safety*, Second edition, International Standardization Organization, Dec. 2018.

[2] J. Raik, H. Fujiwara, R. Ubar, and A. Krivenko, "Untestable fault identification in sequential circuits using model-checking", in *2008 17th Asian Test Symposium*, IEEE, Nov. 2008. DOI: 10.1109/ats.2008.22.

[3] G. Cabodi and M. Murciano, "BDD-based hardware verification", in *Formal Methods for Hardware Verification*, Springer Berlin Heidelberg, 2006, pp. 78–107. DOI: 10.1007/11757283_4.

[4] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny, "Multiway decision graphs for automated hardware verification", *Formal Methods in System Design*, vol. 10, no. 1, pp. 7–46, 1997. DOI: 10.1023/a:1008663530211.

[5] H.-C. Liang, C. L. Lee, and J. Chen, "Identifying untestable faults in sequential circuits", *IEEE Design & Test of Computers*, vol. 12, no. 3, pp. 14–23, 1995. DOI: 10.1109/mdt.1995.466367.

[6] M. Syal and M. Hsiao, "New techniques for untestable fault identification in sequential circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 6, pp. 1117–1131, Jun. 2006. DOI: 10.1109/tcad.2005.855967.

[7] C. Romesburg, *Cluster Analysis for Researchers*. Lulu.com, Apr. 1, 2004, 344 pp., ISBN: 1411606175. [Online]. Available: https://www.ebook.de/de/product/4208765/charles_romesburg_cluster_analysis_for_researchers.html.

[8] S. Mirkhani and Z. Navabi, "Enhancing fault simulation performance by dynamic fault clustering", IEEE, 2005. DOI: 10.1109/ats.2005.58.

[9] A. Evans, M. Nicolaidis, S.-J. Wen, and T. Asis, "Clustering techniques and statistical fault injection for selective mitigation of SEUs in flip-flops", in *International Symposium on Quality Electronic Design (ISQED)*, IEEE, Mar. 2013. DOI: 10.1109/isqed.2013.6523691.

[10] 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)*, IEEE, Jul. 2017. DOI: 10.1109/iolts.2017.8046210.

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

[12] B. Tabacaru, M. Chaari, W. Ecker, C. Novello, T. Kruse, K. Liu, H. Post, N. Hatami, and A. von Schwerin, "Fault-injection techniques for TLM-based virtual prototypes", Sep. 2015.

[13] G. Rodrigues, F. Rosa, A. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the impact of fault tolerance methods in ARM processors under soft errors running linux and parallelization API", *IEEE Transactions on Nuclear Science*, pp. 1–1, 2017. DOI: 10.1109/tns.2017.2706519.

[14] F. Rosa, L. Ost, R. Reis, S. Davidmann, and L. Lapides, "Evaluation of multicore systems soft error reliability using virtual platforms", in *2017 15th IEEE International New Circuits and Systems Conference (NEWCAS)*, IEEE, Jun. 2017. DOI: 10.1109/newcas.2017.8010111.

[15] P. Adelt, B. Koppelmann, W. Mueller, M. Becker, B. Kleinjohann, and C. Scheytt, "Fast dynamic fault injection for virtual microcontroller platforms", in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, Sep. 2016. DOI: 10.1109/vlsi-soc.2016.7753545.

[16] S. Reiter, A. Viehl, O. Bringmann, and W. Rosenstiel, "Fault injection ecosystem for assisted safety validation of automotive systems", in *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, IEEE, Oct. 2016. DOI: 10.1109/hldvt.2016.7748256.

[17] D. Mueller-Gritschneder, P. R. Maier, M. Greim, and U. Schlichtmann, "System C-based multi-level error injection for the evaluation of fault-tolerant systems", in *2014 International Symposium on Integrated Circuits (ISIC)*, IEEE, Dec. 2014. DOI: 10.1109/isicir.2014.7029567.

[18] D. Mueller-Gritschneder, M. Greim, and U. Schlichtmann, "Safety evaluation based on virtual prototypes: Fault injection with multi-level processor models", in *2016 International Symposium on Integrated Circuits (ISIC)*, IEEE, Dec. 2016. DOI: 10.1109/isicir.2016.7829710.

[19] J. Espinosa, C. Hernandez, and J. Abella, "Characterizing fault propagation in safety-critical processor designs", in *2015 IEEE 21st International On-Line Testing Symposium (IOLTS)*, IEEE, Jul. 2015. DOI: 10.1109/iolts.2015.7229848.

[20] J. Espinosa, C. Hernandez, J. Abella, D. de Andres, and J. C. Ruiz, "Analysis and RTL correlation of instruction set simulators for automotive microcontroller robustness verification", in *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, ACM Press, 2015. DOI: 10.1145/2744769.2744798.

[21] F. Corno, P. Prinetto, and M. Sonza Reorda, "Testability analysis and ATPG on behavioral RT-level VHDL", in *Proceedings International Test Conference 1997*, Int. Test Conference, 1997. DOI: 10.1109/test.1997.639688.

[22] S. Ravi, I. Ghosh, V. Boppana, and N. K. Jha, "A technique for identifying RTL and gate-level correspondences", in *Proceedings 2000 International Conference on Computer Design*, IEEE Comput. Soc, 2000. DOI: 10.1109/iccd.2000.878351.

[23] E. Cheung, X. Chen, F. Tsai, Y.-C. Hsu, and H. Hsieh, "Bridging RTL and gate: Correlating different levels of abstraction for design debugging", in *2007 IEEE International High Level Design Validation and Test Workshop*, IEEE, 2007. DOI: 10.1109/hldvt.2007.4392790.

[24] M. Conrad, G. Sandmann, and P. Munier, "Qualifying software tools according to ISO 26262", in *2010 Model-Based Development of Embedded Systems (MBEES)*, 2010. DOI: 10.4271/2011-01-1005.

[25] F. Asplund, "The future of software tool chain safety qualification", *Safety Science*, vol. 74, pp. 37–43, 2015, ISSN: 0925-7535. DOI: 10.1016/j.ssci.2014.11.023.

[26] F. Asplund, J. El-khoury, and M. Törngren, "Qualifying software tools, a systems approach", in *2012 International Conferenceon Computer Safety, Reliability and Security (SAFECOMP)*, 2012, pp. 340–351. DOI: 10.1007/978-3-642-33678-2_29.

[27] Q. Wang, A. Wallin, V. Izosimov, U. Ingelsson, and Z. Peng, "Test tool qualification through fault injection", in *2012 17TH IEEE EUROPEAN TEST SYMPOSIUM (ETS)*, IEEE, May 2012. DOI: 10.1109/ets.2012.6233042.

[28] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Combining fault analysis technologies for ISO26262 functional safety verification", in *2019 IEEE 28th Asian Test Symposium (ATS)*, IEEE, Dec. 2019. DOI: 10.1109/ats47505.2019.00024.

[29] S. Praveen, S. Yellampalli, and A. Kothari, "Optimization of test time and fault grading of functional test vectors using fault simulation flow", in *2014 International Conference on Electronics, Communication and Computational Engineering (ICECCE)*, IEEE, Nov. 2014. DOI: 10.1109/icecce.2014.7086633.

[30] S. Arekapudi, F. Xin, J. Peng, and I. G. Harris, "ATPG for timing-induced functional errors on trigger events in hardware-software systems", in *Proceedings The Seventh IEEE European Test Workshop*, IEEE Comput. Soc, 2002. DOI: 10.1109/etw.2002.1029635.

[31] C. R. Berkeley, "International Workshop on Logic and Synthesis (IWLS) 2005 benchmarks", Tech. Rep., 2005.

[32] *GPDK045 reference manual*, Revision 5.0, Cadence Design Systems , Inc., 2016.

[33] International Standardization Organization, *ISO 26262 Road Vehicles - Function Safety - Part 8: Supporting processes*, Second edition, International Standardization Organization, Dec. 2018.

[34] I. S. Organization, *ISO 26262 Road Vehicles - Function Safety - Part 5: Product development at the hardware level*, Second edition, International Standardization Organization, Dec. 2018.

[35] F. Augusto da Silva, A. C. Bagbaba, A. Ruospo, R. Mariani, G. Kanawati, E. Sanchez, M. Sonza Reorda, M. Jenihhin, S. Hamdioui, and C. Sauer, "Special session: AutoSoC - a suite of open-source automotive SoC benchmarks", in *2020 IEEE 38th VLSI Test Symposium (VTS)*, IEEE, Apr. 2020. DOI: 10.1109/vts48691.2020.9107599.

[36] J. Han, Y. Kwon, Y. C. P. Cho, and H.-J. Yoo, "A 1ghz fault tolerant processor with dynamic lockstep and self-recovering cache for ADAS SoC complying with ISO26262 in automotive electronics", in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, IEEE, Nov. 2017. DOI: 10.1109/asscc.2017.8240279.

[37] A. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing lockstep dual-core ARM cortex-a9 soft error mitigation in freeRTOS applications", in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design Chip on the Sands - SBCCI '17*, ACM Press, 2017. DOI: 10.1145/3109984.3110008.

[38] A. Höller, N. Kajtazovic, T. Rauter, K. Römer, and C. Kreiner, "Evaluation of diverse compiling for software-fault detection", in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, IEEE Conference Publications, 2015. DOI: 10.7873/date.2015.0118.

[39] M. Jenihhin, M. Sonza Reorda, A. Balakrishnan, and D. Alexandrescu, "Challenges of reliability assessment and enhancement in autonomous systems", in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, Oct. 2019. DOI: 10.1109/dft.2019.8875379.

[40] M. Singh and S. Kim, "Security analysis of intelligent vehicles: Challenges and scope", in *2017 International SoC Design Conference (ISOCC)*, IEEE, Nov. 2017. DOI: 10.1109/isocc.2017.8368805.

[41] G. Kalamkar, A. Gotkhindikar, and A. R. Suryawanshi, "Low-level memory attacks on automotive embedded systems", in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, IEEE, Aug. 2018. DOI: 10.1109/iccubea.2018.8697376.

[42] G. Kornaros, O. Tomoutzoglou, and M. Coppola, "Hardware-assisted security in electronic control units: Secure automotive communications by utilizing one-time-programmable network on chip and firewalls", *IEEE Micro*, vol. 38, no. 5, pp. 63–74, Sep. 2018. DOI: 10.1109/mm.2018.053631143.

[43] Renesas, *R-Car M3 Automotve SoC specification*, 2020. [Online]. Available: https://www.renesas.com/us/en/solutions/automotive/soc/r-car-m3.html.

[44] Infineon, *AURIX Family - TC264DA*, 2020. [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/aurix-family-tc264da-adas/.

[45] Texas Instruments, *TDA2SG SoC processor for ADAS applications*, 2020. [Online]. Available: http://www.ti.com/product/TDA2SG.

[46] ISO, *ISO 21434 Road vehicles - Cybersecurity engineering*. [Online]. Available: https://www.iso.org/standard/70918.html.

[47] *Amber 2 core specification*, opencores.org, Mar. 2015.

[48] Cobham Gaisler, *LEON3 multiprocessing CPU core*, 2010.

[49] D. Lampret et al., *OpenRISC 1000 architecture manual*, Revision 0, opencores.org, Dec. 2012.

[50] OpenRISC, *OpenRISC Community*, 2020. [Online]. Available: https://github.com/openrisc.

[51] N. Kanekawa, T. Meguro, K. Isono, Y. Shima, N. Miyazaki, and S. Yamaguchi, "Fault detection and recovery coverage improvement by clock synchronized duplicated systems with optimal time diversity", in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, IEEE Comput. Soc, 1998. DOI: 10.1109/ftcs.1998.689470.

[52] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing", *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, May 2010. DOI: 10.1109/mdt.2010.5.

[53] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers", *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, Mar. 2016. DOI: 10.1109/tc.2015.2498546.

[54] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda, "Test program generation for communication peripherals in processor-based SoC devices", *IEEE Design & Test of Computers*, vol. 26, no. 2, pp. 52–63, Mar. 2009. DOI: 10.1109/mdt.2009.43.

[55] A. Floridia, E. Sanchez, and M. Sonza Reorda, "Fault grading techniques of software test libraries for safety-critical applications", *IEEE Access*, vol. 7, pp. 63 578–63 587, 2019. DOI: 10.1109/access.2019.2917036.

[56] M. Gaudesi, I. Pomeranz, M. Sonza Reorda, and G. Squillero, "New techniques to reduce the execution time of functional test programs", *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1268–1273, Jul. 2017. DOI: 10.1109/tc.2016.2643663.

[57] E. Sanchez, "Increasing reliability of safety critical applications through functional based solutions", in *2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, IEEE, Apr. 2018. DOI: 10.1109/dtis.2018.8368555.

[58] P. D. Schiavone, E. Sanchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An open-source verification framework for open-source cores: A RISC-V case study", in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, Oct. 2018. DOI: 10.1109/vlsi-soc.2018.8644818.

[59] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J.-G. Mess, "Improved test solutions for COTS-based systems in space applications", in *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*, Springer International Publishing, 2019, pp. 187–206. DOI: 10.1007/978-3-030-23425-6_10.

[60] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Flip flop weighting: A technique for estimation of safety metrics in automotive designs", in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, IEEE, Jun. 2021. DOI: 10.1109/iolts52814.2021.9486697.

[61] E. Kang, E. Jackson, and W. Schulte, "An approach for effective design space exploration", in *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, Springer Berlin Heidelberg, 2011, pp. 33–54. DOI: 10.1007/978-3-642-21292-5_3.

[62] S. Chtourou and O. Hammami, "SystemC space exploration of behavioral synthesis options on area, performance and power consumption", in *2005 International Conference on Microelectronics*, IEEE. DOI: 10.1109/icm.2005.1590039.

[63] B. C. Schafer, "Probabilistic multiknob high-level synthesis design space exploration acceleration", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 394–406, Mar. 2016. DOI: 10.1109/tcad.2015.2472007.

[64] K. Roy, H. T. Mert, and M. Swaminathan, "Preliminary application of deep learning to design space exploration", in *2018 IEEE Electrical Design of Advanced Packaging and Systems Symposium (EDAPS)*, IEEE, Dec. 2018. DOI: 10.1109/edaps.2018.8680888.

[65] M. Karunaratne, A. Sagahayroon, and S. Prodhuturi, "RTL fault modeling", in *48th Midwest Symposium on Circuits and Systems, 2005.*, IEEE, 2005. DOI: 10.1109/mwscas.2005.1594451.

[66] M. Siebert, E. Gramatova, and L. Nagy, "PaCGEN: Automatic system for critical path selection based on multiple parameters", in *2014 14th Biennial Baltic Electronic Conference (BEC)*, IEEE, Oct. 2014. DOI: 10.1109/bec.2014.7320565.

[67] H. Fang, K. Chakrabarty, A. Jas, S. Patil, and C. Tirumurti, "RT-level deviation-based grading of functional test sequences", in *2009 27th IEEE VLSI Test Symposium*, IEEE, May 2009. DOI: 10.1109/vts.2009.12.

[68] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J.-G. Mess, "An analysis of test solutions for COTS-based systems in space applications", in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, IEEE, Oct. 2018. DOI: 10.1109/vlsi-soc.2018.8644846.

[69] ARM, *Development tools and software - Software Test Libraries*, 2019.

[70] Cypress, *AN204377 FM3 and FM4 family, IEC61508 SIL2 self-test library*, 2017.

[71] Microchip, *DS52076A 16-bit CPU self-test library user's guide*, 2012.

[72] F. Augusto da Silva, A. C. Bagbaba, 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)*, IEEE, Oct. 2021. DOI: 10.1109/itc50571.2021.00047.

[73] 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*, IEEE, 2014. DOI: 10.1109/vlsi-dat.2014.6834876.

CURRICULUM VITÆ

Felipe AUGUSTO DA SILVA

PERSONAL INFO

14-10-1986	Date of Birth in São Paulo, Brazil.
E-mail	felipeaugdasilva@gmail.com

EDUCATION

2018–2022 Ph.D. degree in Computer Engineering
Delft University of Technology (TU Delft), the Netherlands
Thesis: EDA tools and methodologies for reliable nanoelectronic systems
Supervisors: Prof. dr. ir. S. Hamdioui (TU Delft)
Dr. C. Sauer (Cadence)

2010–2014 M.Sc. degree in Electrical and Electronics Engineering
Federal University of Santa Catarina (UFSC), Brazil
Thesis: Conception and Validation of a robust architecture based on Soft Processors aiming On Board Computers for artificial Satellites
Supervisor: Prof. dr. E. Bezerra (UFSC)

2004–2009 Engineer's degree in Computer Engineering
Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil

EXPERIENCE

2017–Present	Functional Safety Specialist, Cadence Design Systems, Germany
2015–2017	Avionics Software Team Leader, AEL Sistemas (Elbit Systems Group), Brazil
2014–2017	Technical Manager, AEL Sistemas (Elbit Systems Group), Brazil
2012–2014	Software Engineer, Elbit Systems, Israel

List of Publications

12. **F. Augusto da Silva**, R. Cantoro, S. Hamdioui, S. Sartoni, C. Sauer, M. Sonza Reorda, "A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers," Electronics, **MDPI** AG, 2022, Vol. 11, No. 16, doi:10.3390/electronics11162481

11. J. E. Rodriguez Condia, **F. Augusto da Silva**, A. Bagbaba, J. D. Guerrero-Balaguera, S. Hamdioui, C. Sauer, M. Sonza Reorda, "Using STLs for Effective In-field Test of GPUs," **IEEE Design & Test**, 2022 (early-access), doi:10.1109/MDAT.2022.3188573

10. A. Bagbaba, **F. Augusto da Silva**, M. Sonza Reorda, S. Hamdioui, M. Jenihhin, C. Sauer, "Automated Identification of Application-Dependent Safe Faults in Automotive Systems-on-a-Chips," Electronics, **MDPI** AG, 2022, 11, 319, doi:10.3390/electronics11030319

9. **F. Augusto da Silva**, A. Bagbaba, S. Hamdioui, C. Sauer, "An automated formal-based approach for reducing undetected faults in ISO 26262 hardware compliant designs," 2021 IEEE International Test Conference (**ITC**), IEEE, 2021, doi:10.1109/itc50571.2021.00047

8. **F. Augusto da Silva**, A. Bagbaba, S. Hamdioui, C. Sauer, "Flip Flop Weighting: A technique for estimation of safety metrics in Automotive Designs," 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (**IOLTS**), IEEE, 2021, doi:10.1109/iolts52814.2021.9486697

7. **F. Augusto da Silva**, A. Bagbaba, S. Sartoni, R. Cantoro, M. Sonza Reorda, S. Hamdioui, C. Sauer, "Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs 2020 IEEE European Test Symposium (**ETS**)," IEEE, 2020, doi:10.1109/ets48528.2020.9131568

6. **F. Augusto da Silva**, A. Bagbaba, A. Ruospo, R. Mariani, G. Kanawati, E. Sanchez, M. Sonza Reorda, M. Jenihhin, S. Hamdioui, C. Sauer, "Special Session: AutoSoC - A Suite of Open-Source Automotive SoC Benchmarks," 2020 IEEE 38th VLSI Test Symposium (**VTS**), IEEE, 2020, doi:10.1109/vts48691.2020.9107599

5. J. E. R. Condia, **F. Augusto da Silva**, S. Hamdioui, C. Sauer, M. Sonza Reorda, "Untestable faults identification in GPGPUs for safety-critical applications," 2019 26th IEEE International Conference on Electronics, Circuits and Systems (**ICECS**), IEEE, 2019, doi:10.1109/icecs46596.2019.8964677

4. **F. Augusto da Silva**, A. Bagbaba, S. Hamdioui, C. Sauer, "Combining Fault Analysis Technologies for ISO26262 Functional Safety Verification," 2019 IEEE 28th Asian Test Symposium (**ATS**), IEEE, 2019, doi:10.1109/ats47505.2019.00024

3. **F. Augusto da Silva**, A. Bagbaba, S. Hamdioui, C. Sauer, "Efficient Methodology for ISO26262 Functional Safety Verification," 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (**IOLTS**), IEEE, 2019, doi:10.1109/iolts.2019.8854449

2. **F. Augusto da Silva**, A. Bagbaba, C. Sauer, "Use of Formal Methods for verification and optimization of Fault Lists in the scope of ISO26262," 2018 Design and Verification Conference (**DVCon**) Europe, Zenodo, 2018, doi:10.5281/zenodo.3361533

1. A. Bagbaba, **F. Augusto da Silva**, C. Sauer, "Improving the Confidence Level in Functional Safety Simulation Tools for ISO26262," 2018 Design and Verification Conference (**DVCon**) Europe, Zenodo, 2018 doi:10.5281/zenodo.3361607