DOCTORAL THESIS

Methods to Optimize Functional Safety Assessment for Automotive Integrated Circuits

Ahmet Çağrı Bağbaba

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2022

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
9/2022

Methods to Optimize Functional Safety Assessment for Automotive Integrated Circuits

AHMET ÇAĞRI BAĞBABA

TALTECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Computer Systems

The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer and Systems Engineering on 4 March 2022

Defence of the thesis: 20 April 2022, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Ahmet Çağrı Bağbaba ______________________
signature

Copyright: Ahmet Çağrı Bağbaba, 2022
ISSN 2585-6898 (publication)
ISBN 978-9949-83-802-8 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-803-5 (PDF)
Printed by Koopia Niini & Rauam

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
9/2022

Meetodid autotööstuse kiipide funktsionaalse ohutuse hindamise optimeerimiseks

AHMET ÇAĞRI BAĞBABA

TALTECH
KIRJASTUS

Contents

List of Publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I A. C. Bagbaba, F. Augusto da Silva, and C. Sauer. Improving the confidence level in functional safety simulation tools for iso 26262. In *2018 Design and Verification Conference (DVCON)*, 2018

II F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer. Use of formal methods for verification and optimization of fault lists in the scope of iso26262. In *2018 Design and Verification Conference (DVCON)*, 2018

III Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Efficient methodology for iso26262 functional safety verification. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 255–256, 2019

IV Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Efficient fault injection based on dynamic hdl slicing technique. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 52–53, 2019

V Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 129–1295, 2019

VI Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Accelerating transient fault injection campaigns by using dynamic hdl slicing. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019

VII Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Annachiara Ruospo, Riccardo Mariani, Ghani Kanawati, Ernesto Sanchez, Matteo Sonza Reorda, Maksim Jenihhin, Said Hamdioui, and Christian Sauer. Special session: Autosoc - a suite of open-source automotive soc benchmarks. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–9, 2020

VIII Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Sandro Sartoni, Riccardo Cantoro, Matteo Sonza Reorda, Said Hamdioui, and Christian Sauer. Determined-safe faults identification: A step towards iso26262 hardware compliant designs. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020

IX Ahmet Cagri Bagbaba, Maksim Jenihhin, Raimund Ubar, and Christian Sauer. Representing gate-level set faults by multiple seu faults at rtl. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2020

X Ahmet Cagri Bagbaba, Felipe Augusto da Silva, Matteo Sonza Reorda, Said Hamdioui, Maksim Jenihhin, and Christian Sauer. Automated identification of application-dependent safe faults in automotive systems-on-a-chips. *Electronics*, 11(3), 2022

Author’s Contributions to the Publications

I In I, the author wrote the scripts for simulation-based fault injection and ATPG flows. The author also developed the idea and the application that uses ATPG test benches and test vectors in fault injection campaigns. Additionally, the author carried out all the simulations and collected the results.

II In II, the author contributed during the development of the presented idea. The author assisted to writing.

III In III, the author supported the combination of ATPG and fault injection flows with the scripts he developed. The author also synthesized and tested the designs that were used to carry out experiments.

IV In IV, the author carried out the research on dynamic slicing. The author wrote all the scripts, implemented the idea using the required tools, prepared the target designs, conducted experiments, wrote the manuscript, and presented it at the conference.

V In V, which is the extension of III, the author advanced the use of ATPG and fault injection to obtain better safety results. Similar to the previous one, the author also synthesized and tested the designs. The author also helped to elaborate the manuscript.

VI In VI, the author extended and improved the method presented in the previous paper on dynamic slicing. The author also integrated the new algorithm into the tools, handled the experiments, wrote the manuscript, and presented it at the conference.

VII In VII, the author was involved in developing the idea, designed the gate-level representation of the presented benchmark suite, ran simulations and fault injection campaigns using special test programs.

VIII In VIII, the author contributed to the gate-level implementation of the target design, ran the tests, and carried out fault injection campaigns.

IX In IX, the author developed the concept that maps faults from gate-level to RTL abstraction level. The author wrote all the scripts, ran the simulations, collected the experimental results, wrote the paper, and presented in the conference.

X In X, the author established the idea of application-specific safe faults on an automotive system-on-chip. The author developed the automotive representative software application, wrote the scripts to characterize it, automatically generate formal properties, and ran a formal analysis to increase the safety level of the target design. The author also wrote the manuscript.

Abbreviations

ACC	Adaptive Cruise Controller
ADAS	Advanced Driver Assistance System
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIL	Automotive Safety Integrity Level
ATPG	Automatic Test Pattern Generation
BIST	Built-in-Self-Test
CAN	Controller Area Network
CCA	Cruise-Control-Application
CRC	Cyclic Redundancy Check
CPU	Central Processing Unit
DC	Diagnostic Coverage
DCLS	Dual-Core Lockstep
DFA	Dependent-Failure Analysis
DMR	Dual Modular Redundancy
DfT	Design-for-Test
EBS	Emergency Braking System
ECC	Error Correction Code
ECU	Electronic Control Unit
EDA	Electronic Design Automation
FIFO	First-in-First-out
FPGA	Field-Programmable Gate Array
FTA	Fault-tree Analysis
FMEDA	Failure Modes, Effects, and Diagnostics Analysis
GPDK	Generic Process Design Kit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Property
ISO	International Organization for Standardization
JTAG	Joint Test Action Group
RAM	Random-Access Memory
RTL	Register Transfer Level
SBST	Software-based Self Test
SA	Stuck-at
SER	Soft Error Reliability
SET	Single Event Transient
SEU	Single Event Upset
SoC	System-on-Chip
STL	Software Test Library
TCL	Tool Confidence Level
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver-Transmitter
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLSI	Very Large-Scale Integration

1 Introduction

This PhD thesis addresses functional safety assessment methods targeting automotive Integrated Circuits (IC), emphasizing optimization techniques.

This introductory chapter presents the motivation behind this thesis, formulates the area's problems, lists a summary of the main contributions, and sketches the thesis structure.

1.1 Motivation

In recent years, the use of Systems-on-Chip (SoCs) in automotive has been increasing rapidly. The reason for this is the Advanced Driver-Assistance Systems (ADAS) used in cars or the addition of autonomous-driving features, which is the most popular topic in automotive domain today. [11] reports that today's cars consist of more than 100 electronic control units (ECUs) to handle different applications. ADAS systems such as Adaptive Cruise Controller (ACC) or Emergency Braking Systems (EBS), which can be life-threatening in case of any malfunction, are present in every car today. For this reason, it is crucial that SoCs used in areas where safety is of vital importance, such as automotive or space, operate without failing. Otherwise, it may cause injury or even loss of human lives. Consequently, International Organization for Standardization (ISO) has developed ISO 26262 [12] functional safety standard in 2011 and revised in 2018 targeting design, verification, and validation of safety-critical automotive SoCs against systematic and random faults.

Several reasons can cause a failure in an SoC. First, it is the decreasing trend in the transistor size, which is described by "Moore's Law" [13]. When the advanced technology nodes take place in the design of an SoC, random hardware faults become more effective as shrinking nodes increase the sensitivity of SoCs to aging effects [14] [15] or cosmic radiation [16]. Therefore, as safety-critical SoCs are designed using advanced node technologies, the need for more safe SoCs is increasing accordingly. Second, it is the increasing trend in the density of complex electronic devices inside automotive SoCs. Modern cars have several advanced features to become more autonomous that require the processing of real-life data during driving, monitoring the environment, and analyzing traffic signs. All these tasks increase the SoC density in the cars. Additionally, [17] states that ninety percent of the car novelties are based on electronics, proving how complexity in car electronics is growing. Because of high-density electronics that manage the above-mentioned advanced car features, SoCs in automobiles are becoming more susceptible to random hardware faults that can cause data loss and harm human health. As a result, there is a need to design robust and safe SoCs to avoid life-critical failures.

The way of preventing failures in SoCs is to implement fault prevention mechanisms (safety mechanisms) that detect faults or control failures to maintain or achieve a safe state [12] [18]. Safety mechanisms are introduced during the design and verification of the hardware/software system. Examples of safety mechanisms include Built-in-Self-Test (BIST), Error Correction Codes (ECC), or hardware redundancy. However, safety mechanisms increase the design and verification complexity as well as the cost of safety-critical SoCs. For example, one of the widely used hardware redundancy schemes is Dual-Core Lock Step (DCLS) increases the SoC's area and power consumption, whereas some other safety mechanisms have an adverse impact on the SoC performance [19] [20]. Moreover, ISO 26262, the functional safety standard, requires using safety analysis techniques to measure the effectiveness and evaluate the benefits of the implemented safety mechanisms. This necessity also escalates the verification complexity of a safety-critical SoC development flow and makes it difficult to comply with the time-to-market criteria. Therefore, methods to overcome the complexity are crucial.

In this PhD thesis, we address functional safety analysis of Electrical/Electronic (E/E) systems, particularly SoCs used in automobiles. Its particular significance in industrial applications, including automotive, is the fundamental motivation of this thesis. Additionally, existing techniques adopted by the industry and academia still have considerable drawbacks that leave functional safety assessment a challenging and expensive task. Therefore, this thesis proposes solutions to optimize these techniques such as fault injection, formal analysis, or Automatic Test Pattern Generation (ATPG). In addition to the fact that these mentioned methods are handled separately in the following chapters, how their combination can contribute to functional safety assessment has motivated this thesis.

1.2 Problem Formulation

Considering all the constraints introduced above, it is mandatory to improve the efficiency of safety analysis techniques to meet the demanding requirements of the automotive industry. ISO 26262 guides functional safety for the SoCs used in automobiles and recommends using some safety analysis techniques. The purpose is to provide evidence that the SoCs can handle faults through their safety mechanisms, and the functionality of the SoC cannot be changed. Even though helpful safety analysis techniques exist in the literature, there is still room for more efficient and optimized methodologies to deal with different functional safety challenges. To be more exact, these safety analysis techniques need optimization, automation, and being more accurate. Therefore, this PhD thesis proposes techniques to make functional safety analysis of automotive SoCs more effective. The following paragraphs formulate the problems in the area.

Functional safety analysis is a complex procedure that requires challenging skills as follows. First, it necessitates prior knowledge about the application scenarios and the target design. Second, it demands to identify possible safety threats, predict their impacts on the results, add safety mechanisms to prevent failures, and evaluate the severity of the faults. In general, these activities happen before the actual design and implementation are made available. Also, this analysis is performed manually based on expert judgment. However, this is error-prone, time-consuming, thus expensive considering the size and complexity of automotive SoCs. Therefore, the manual analysis is tedious, inefficient, and might have an adverse impact on the SoC development cycle. More specifically, when we take fault injection or formal analysis campaigns into account, or when they are used together, it is impossible to have a complete run considering the complexity of designs and applications. These kinds of functional safety campaigns leave a considerable amount of undetected faults, whose effects are unknown but must be identified by experts manually. Consequently, the lack of automated safety approaches to reduce possible human errors and meet the demanding safety requirements of today's cars is a significant problem.

Moreover, along with the design development, the initially defined requirements must be repeatedly observed while a detailed analysis is performed to provide evidence that these safety requirements are achieved. Nevertheless, the size and complexity of SoCs pose a severe problem in this step, and all functional safety analysis techniques such as fault injection or formal analysis suffer from this issue. In other words, modern automotive SoCs consist of large amounts of faults up to millions; hence, the execution of safety analysis methods on these SoCs requires a long time to obtain significant statistical results. As a result, the industry adopted the "divide and conquer" approach, which is mapping the requirements to the corresponding design components to enable a safety analysis on only safety-related design parts [21]. However, even with this solution, functional safety analysis campaigns are more costly and even unfeasible considering complex ECUs that perform various application scenarios that automate the car [22]. This also shows that not only hardware complexity but also real-world applications that SoCs execute need to be carefully considered for efficient functional safety analysis. On the other hand, characterization of these applications (workloads) is a challenge and needs to be carefully performed to develop optimized methods and techniques that speed up functional safety analysis and reduce the cost of the SoC design and verification cycle.

Also, the development of test environments that runs functional safety analysis is another problem. The test environment is necessary to obtain a confident assessment of results generated by the tools in the context of ISO 26262. This implies that compelling methodologies that reduce the efforts of test environment development are necessary. Furthermore, the attempts to unify functional safety analysis tools/technologies and their strengths create a competent campaign, making all kinds of analysis and their results more confident. Hence, it is required to have this environment supporting ISO 26262 compliant automotive SoC development.

Additionally, there is a lack of open-source resources in the functional safety research community. The high need to evaluate the quality of results on automotive representative SoCs makes any research on the area more challenging. Moreover, both hardware and software resources must be available to demonstrate various use cases. Also, providing a representative automotive SoC that has several safety mechanisms capable of detecting random faults is essential. Nevertheless, it is a problem to access this kind of comprehensive automotive SoC with a modular structure, enabling more detailed research.

1.3 Research Objectives

Considering the above problems, the main goal of this PhD thesis is based on addressing them in the following ways:

- Proposing a novel methodology that combines different fault analysis tools. The idea is to increase tool confidence level as guided by ISO 26262.
- Providing an open-sourced benchmark to support functional safety research, enabling comprehensive and automotive representative SoC with all required hardware and software resources.
- Optimizing fault injection campaigns by the analysis of workloads that run on the target designs. The main idea is to select critical faults to be injected and so prune the large fault lists.
- Enhancing hardware fault classification, which is normally done manually based on expert judgment. Analysis of the software application (workload) automatically and translation of this behavior to the formal properties accurately are aimed.

1.4 Contributions

The main objective of this PhD thesis is to propose methodologies to leverage the most advanced functional safety analysis techniques. Contributions of this PhD thesis in tackling the challenges explained in Chapter 1.2 consist;

- Mitigation of the drawbacks caused by the different classification characteristics of fault analysis tools is addressed in this PhD thesis. Also, the applicability and feasibility of these tools are investigated.

Taking the advantage of the strength of one specific fault analysis tool as a starting point, fault injection campaigns are optimized, providing better coverage.

Unified application of three tools, processing and report generation, are the main features.

More details about this approach is clarified in Chapter 3.

- Filling the gap in the research community by proposing representative automotive SoC is studied in the context of this thesis. The requirements of an automotive SoC are examined by inspecting several commercial alternatives.

  The approach shall be oriented towards seamless hardware and software resources, modular SoC structure enabling adding peripherals, different safety configurations in compliance with ISO 26262.

  The ultimate goal of this work is to build a representative automotive SoC, supporting functional safety researches, which needs an open-source platform to assess the quality of the results.

  Chapter 4 provides the details of the proposed approach.

- Accelerating fault injection campaigns targeting transient faults is investigated in the context of this thesis. Starting with a state-of-the-art study of the main fault injection acceleration techniques applied in the industry and academia, efficient methodologies are developed to prune the fault lists of transient fault injection campaigns.

  The main objective of this approach is to initiate a general and well-structured description of fault injection campaigns, in particular simulation-based fault injection, which supports industrial-grade automotive SoCs and applications.

  A key milestone in the proposed approach is the workload characterization and identification of critical and non-critical time steps, mapping faults between abstraction levels, and pruning the transient fault list accordingly.

  More detailed explanations are given in Chapter 5.

- Enhancing hardware fault classification using formal analysis is investigated in this thesis. Starting with an analysis of software application and development of formal properties, safe fault identification is intensified.

  The main aim of this research is to an analysis of a complete automotive SoC when it runs an automotive representative application software. Furthermore, characterization of the faults on peripherals is also targeted when the application software uses them.

  Automated formal property generation, processing, and combination of formal analysis and fault injection to achieve safety metrics driven by ISO 26262 are considered.

  The results and details for enhancing the hardware fault classification approach are elicited in Chapter 6.

1.5 Thesis Organization

This thesis consists of seven main chapters. The rest of it is organized as follows.

- Chapter 2 provides a background information about functional safety and ISO 26262 standard to familiarize the reader with the basics.

- Chapter 3 points out combination of fault analysis technologies to increase tool's confidence level, and have a better detection rate using the strength of the ATPG.
- Chapter 4 defines a open-source and comprehensive automotive SoC, the AutoSoC, to support research targeting functional safety.
- Chapter 5 starts with the discussion about the challenges of fault injection campaigns and then provides a solution to prune the fault lists.
- Chapter 6 explains enhancing hardware fault classification in an automotive SoC using formal methods.
- Chapter 7 draws conclusions for the presented PhD thesis and discusses possible future directions in the functional safety scope.

Also, at the end of the thesis, the research papers mentioned in the context of this PhD are attached as appendixes.

2 Background on Functional Safety and ISO 26262

This chapter initially overviews the functional safety and its usage areas. Then, automotive functional safety is detailed further to give the reader more information about the main scope of the presented research.

2.1 Functional Safety: A General Overview

In general, functional safety refers to the concept that a system (typically an SoC, or an IP) will remain dependable and function as intended even in an unexpected occurrence, which is termed a failure. Failures can be due to random hardware faults (e.g., short circuits) or systematic design errors (e.g., defects in software). Furthermore, the possible risk posed by these failures must be reduced by either minimizing the probability of a failure occurring or restricting the aftereffects of unavoidable failures. Concerning safety-critical areas such as automotive, planes, or medical devices, embedded electronics have increased significantly, and therefore, there is an increased emphasis on functional safety in the designs used in these areas.

The use of advanced node technologies that are adopted increasingly for performance and reduced area/power escalates susceptibility of SoCs to radiation sources and aging effects. However, these two impacts cause the device to malfunction temporarily or permanently. Moreover, malfunctioning of an SoC used in safety-critical areas might pose a significant risk even for a short period. Consequently, several standards have been developed to make the system safe.

The following sub-chapter explains application areas and relates standards developed for safety-critical areas.

2.1.1 Application Areas and Related Standards

IEC 61508 [23] is the international standard that provides generic guidelines for the specification, design, and operation of Electric/Electronic and Programmable Electronic (E/E/PE) systems used in safety-critical areas. It is a generic safety standard and serves as the basis for drafting the functional safety guidelines tailored to the respective industry sectors. Fig. 1 gives an overview of functional safety standards. This illustration explains that IEC 61508 supported the development of safety standards applied to process industry, nuclear, medical, machinery, aviation, automotive, and many others that are not included in Fig. 1.

Among these standards, ISO 26262 [12] "Road Vehicles - Functional Safety" has been published in 2011, targeting the automotive industry. This standard addresses series production passenger cars up to 3500 kilograms. Moreover, ISO 26262 [12] targets the high-volume mass-market automotive industry, whereas IEC 61508 deals with the systems produced in low volumes. In this thesis, the focus is on the ISO 26262 functional safety standard. Hence, the following sub-chapters give a detailed study of the standard.

2.2 Automotive Functional Safety: ISO 26262

Being the adaptation of IEC 61508, ISO 26262 is a functional safety standard titled "Road Vehicles - Functional Safety". It was first published in 2011 as a functional safety standard for the automotive industry. Then, its revised second edition was released in 2018. This second edition has broader scope by removing the vehicle mass limitation mentioned in Chapter 2.1.1 and also includes two additional chapters for guidelines on the application of ISO 26262 to semiconductors and the adaptation for motorcycles.

According to the ISO 26262, functional safety is described as the "absence of unrea-

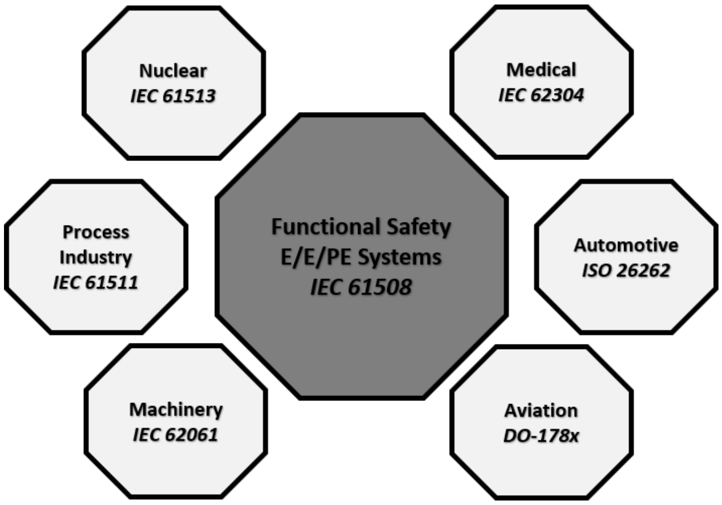


Figure 1: Overview of Functional Safety Standards in Different Industry Sectors

sonable risk due to hazards caused by malfunctioning behavior of electrical/electronic systems". This definition shows that functional safety in automobiles is actually a chain of implications [18] as presented in Fig. 2. It starts with the malfunction definition and targets risk reduction as a final goal.

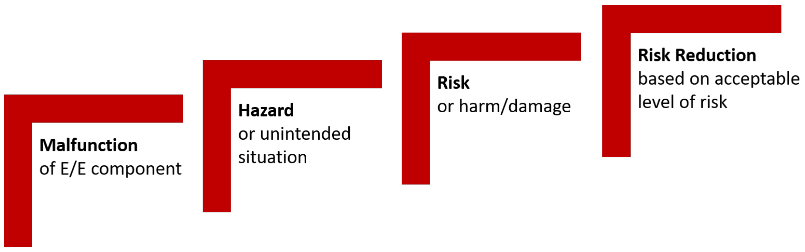


Figure 2: ISO 26262 Chain of Implications [18]

In general, ISO 26262 performs the following tasks:

- It guides to avoid risk in creating a safety-critical system.
- It regulates critical testing processes.
- It introduces Automotive Safety Integrity Levels (ASILs) to specify the item's necessary safety requirements to achieve an acceptable residual risk.

ISO 26262 is based on a V-model, shown in Fig. 3, as a reference process for the different phases of product development. ISO 26262 contains twelve parts covering all required activities to ensure the functional safety of E/E components used in automobiles. These are summarized as follows:

- In Part 1: *Vocabulary*, necessary terms are introduced such as ASIL, item, system, element, and many others.

- In Part 2: *Management of Functional Safety*, the process for management of functional safety for automotive applications is outlined, and the automotive safety lifecycle is introduced.
- In Part 3: *Concept Phase*, activities to be performed are specified. These activities are Item Definition, Hazard and Risk Analysis (HARA), Safety Goals Definition, ASIL Determination, and definition of Functional Safety Requirements. This part is applied during the early phase of product development.
- In Part 4: *Product Development at the System Level*, Technical Safety Requirements are derived for each Functional Safety Requirement defined in the previous chapter with respect to hardware and software components.
- In Part 5: *Product Development at the Hardware Level*, requirements for product development on the hardware level are defined. This part covers hardware design and evaluation of architectural hardware metrics. Also, an assessment of safety goal violation because of random failures is performed.
- In Part 6: *Product Development at the Software Level*, specifications for software safety are defined in this part. Additionally, qualitative analyses, like Failure Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA), are used in the context of this part.
- In Part 7: *Production and Operation*, requirements for system production, operation, installation, servicing, decommissioning are specified.
- In Part 8: *Supporting Processes*, requirements for processes that support the development effort, including documentation standards, tool qualification, verification, and validation, are mentioned.
- In Part 9: *Automotive Safety Integrity Level-Oriented and Safety-Oriented Analyses*, all aspects regarding the ASIL-oriented requirements are explained. Analysis of dependent failures is also covered.
- In Part 10: *Guideline on ISO 26262*, an overview of the standard is summarized to improve the understanding of other parts.
- In Part 11: *Guideline on Application of ISO 26262 to semiconductors*, information to support semiconductor manufacturers and silicon intellectual property is provided to address how suppliers and integrators work together.
- In Part 12: *Adaptation of ISO 26262 for Motorcycles*, standard is specified for motorcycles.

2.3 Failure Types

ISO 26262 classifies malfunction of E/E component into *Systematic Failures*, *Random Hardware Failures*, and *Dependent Failures*. This section discusses these three types of failures in detail.

- **Systematic Failures**: According to Part 1 of the standard [12], these are the failures in an item or function that are caused in a deterministic way during development, manufacturing, or maintenance. Systematic failures can only be eliminated by changing the design or the manufacturing process, operational procedures, or other relevant factors.

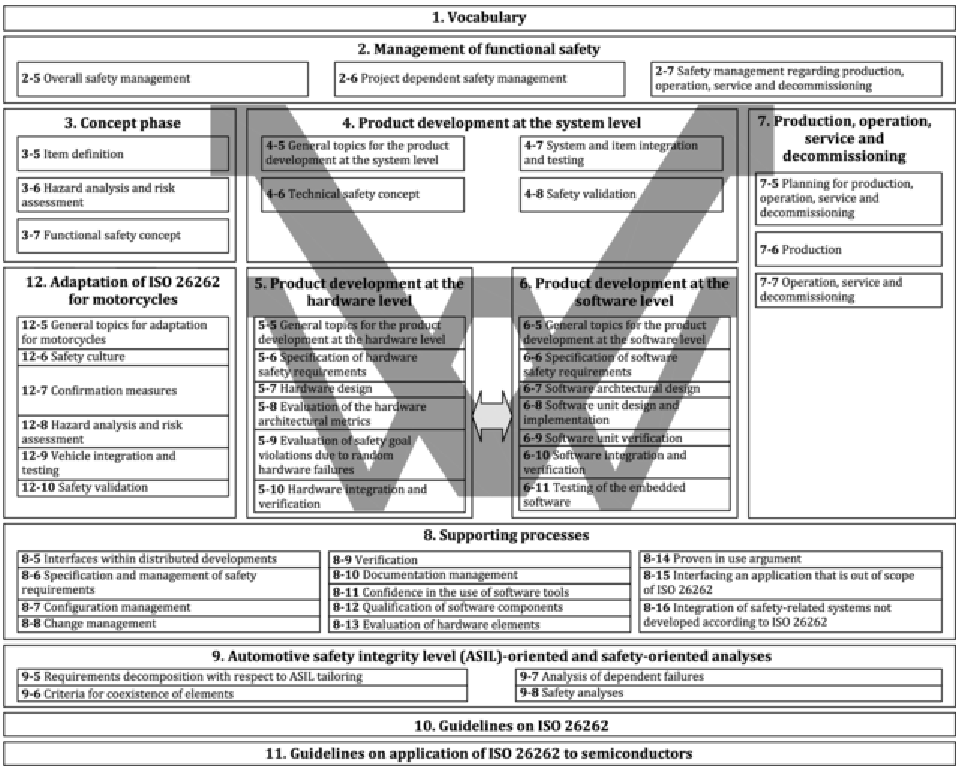

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Project dependent safety management
2-7 Safety management regarding production, operation, service and decommissioning
3. Concept phase
3-5 Item definition
3-6 Hazard analysis and risk assessment
3-7 Functional safety concept
4. Product development at the system level
4-5 General topics for the product development at the system level
4-7 System and item integration and testing
4-6 Technical safety concept
4-8 Safety validation
7. Production, operation, service and decommissioning
7-5 Planning for production, operation, service and decommissioning
7-6 Production
7-7 Operation, service and decommissioning
12. Adaptation of ISO 26262 for motorcycles
12-5 General topics for adaptation for motorcycles
12-6 Safety culture
12-7 Confirmation measures
12-8 Hazard analysis and risk assessment
12-9 Vehicle integration and testing
12-10 Safety validation
5. Product development at the hardware level
5-5 General topics for the product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of safety goal violations due to random hardware failures
5-10 Hardware integration and verification
6. Product development at the software level
6-5 General topics for the product development at the software level
6-6 Specification of software safety requirements
6-7 Software archtectural design
6-8 Software unit design and implementation
6-9 Software unit verification
6-10 Software integration and verification
6-11 Testing of the embedded software
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation management
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Evaluation of hardware elements
8-14 Proven in use argument
8-15 Interfacing an application that is out of scope of ISO 26262
8-16 Integration of safety-related systems not developed according to ISO 26262
9. Automotive safety integrity level (ASIL)-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guidelines on ISO 26262
11. Guidelines on application of ISO 26262 to semiconductors

Figure 3: General Overview of ISO 26262 [12]

- **Random Hardware Failures**: These failures occur unpredictably during the lifetime of a hardware element, and that follows a probability distribution. To prevent Random Hardware Failures, it is necessary to deploy safety mechanisms to make the architecture able to detect and correct malfunctions.

- **Dependent Failures**: These failures are defined as the failure of more than one element stemming from a single root. The main reason for these failures is environmental conditions, aging, or failures of mutual external sources such as power supply. They can be curtailed by supervision of clock, power, temperature, and independent failure signaling [11].

Concerning Random Hardware Failures, Fig. 4 shows that they can be caused by three types of faults based on their duration. These are Intermittent, Permanent, and Transient faults that are explained below:

- **Intermittent Faults**: These are faults in a hardware element that appear, disappear, and then reappear after some time. In other words, an intermittent fault occurs at intervals, usually irregular, in a target system that functions normally at other times. They are caused by poor solder joints, corrosion on connector contacts. Time-dependent alternations in hardware are an example of this type of fault. Thus, these are temporary faults.

- **Transient Faults**: A Transient fault is a malfunction of a device or system that remains active for a short period of time with respect to the device or system mission time. A well-known example of a transient fault is a soft error that hits a device's memory elements and flips the bit from 0 to 1 or 1 to 0. If this happens, the state of a few bits is changed; however, there is no lasting damage to the device.

- **Permanent Faults**: Permanent faults occur and stay until corrective action is taken. Burn-out hardware or disk head crashes are an example of permanent faults.

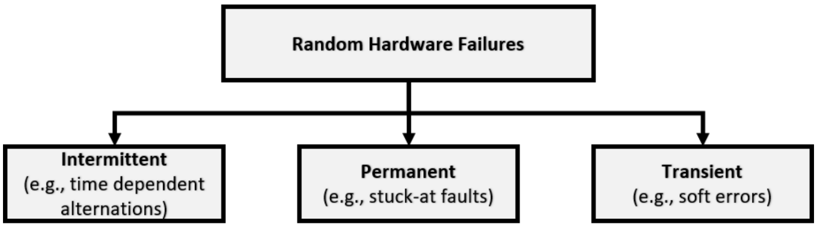


Figure 4: Random Hardware Failure Causes

2.3.1 Fault Models

In order to analyze the effect of hardware faults listed above, fault models must be used on the target design. Even if fault models are not perfect representations of what is happening on a real design, they help design test cases or procedures to mimic and simulate faulty conditions and develop safety mechanisms. In this way, safety-critical designs can be tested whether safety requirements are met even if some hardware components fail or not.

There are two fault models adopted by the industry as listed and explained below:

- **Stuck-at Fault Model**: The stuck-at (SA) model is widely used to address the test of ICs. It is a particular fault model used by ATPG tools to mimic a manufacturing defect within an IC.

  The model assumes that a signal is forced to either 0 (SA0) or 1 (SA1). Thus, it can be applied to any signal, such as nets or registers.

- **Single-Event-Upset (SEU) and Single-Event-Transient (SET) Fault Models**: A SEU is a change of state caused by ionizing particles such as electrons or photons that strike a sensitive node in a device. The model inverts the value of a sequential element's output and holds the modified value until it is assigned a new value. It is applied only on the outputs of sequential components such as flip-flops, latches, or memories.

  The SET fault model inverts the value of a signal and holds the value for a specified period of time. It can be employed to any kind of signal, such as nets or registers, as opposed to SEU.

2.4 Safety Mechanisms

Safety mechanisms should be introduced to address random hardware failures. According to the ISO 26262: Part 1 [12], a safety mechanism is a technical solution implemented by E/E functions or elements, or by other technologies, to detect faults or control failures to achieve or maintain a safe state.

In other words, safety mechanisms are protection mechanisms. They can be implemented using hardware or software techniques. Fig. 5 illustrates an example SoC that

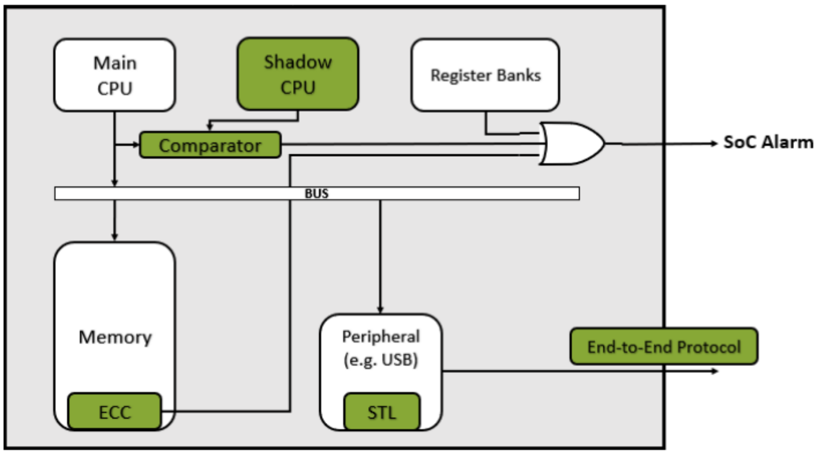


Figure 5: Example Safety Mechanisms in an SoC

has several safety mechanisms. In this example, some major hardware components are protected as follows:

- **CPU**: A shadow CPU, which is the copy of the main, and a comparator are used as a hardware safety mechanism to protect the CPU. This is a typical implementation of DCLS.
- **Memory**: The memory in the example has ECC as a hardware safety mechanism.
- **USB**: Software Test Library (STL) is shown as a software safety mechanism that protects USB. Also, End-to-End Protocol can be used for the transferred data.
- **SoC Alarm**: This safety mechanism works when one of the safety mechanisms mentioned above sets the alarm. In this case, the system is switched to a safe state.

The safety mechanisms shown in Fig. 5 are selected as they are relevant to this PhD thesis. Besides the listed mechanisms above, there are other well-known safety mechanism implementations such as watchdog (program sequence monitor), hardware logic BIST, or ECC mechanisms on instruction cache, data cache, buses, and others. In addition, ISO 26262: Part 5 proposes several categories of safety mechanisms targeting processing units and volatile memory with typical achievable Diagnostic Coverage (DC).

Moreover, safety mechanism selection must be carried out carefully, considering the trade-offs between effectiveness and cost. This selection must evaluate power consumption, area, safety metrics, and timing performance all together for a specific building block [24]. For example, DCLS and Triple Modular Redundancy (TMR) have high area overhead and low-performance impact. On the other hand, a simple parity safety mechanism has lower area overhead and performance impact than DCLS and TMR.

2.5 Functional Safety Analysis

Functional safety analysis is utilized to assess the target product's safety level, which is typically an IP or an SoC. In general, these analysis techniques are grouped into quantitative evaluations and qualitative assessments. Quantitative evaluations are Failure Mode

Effect and Diagnostic Analysis (FMEDA) or timing analysis. On the other hand, Dependent Failure Analysis (DFA) is a well-known example of qualitative assessment.

The following sub-chapter exemplifies the quantitative assessment following the structure of FMEDA.

2.5.1 Quantitative Assessment of Hardware Architectures

The incorporation of several protection mechanisms in a safety-critical design requires the evaluation of their safety level. ISO 26262 addresses this issue through quantitative assessments to determine a hardware architecture's Automotive Safety Integrity Level (ASIL). Four ASILs are defined in ISO 26262, where ASIL-A represents the least stringent level, and ASIL-D is the most stringent level [12]. In order to decide the ASIL of a target design, hardware architectural metrics are designated in ISO 26262. Hardware architectural metrics measure the effectiveness of safety mechanisms to detect random hardware failures. In other words, these metrics assess the overall likelihood of risk.

The hardware architecture assessment flow is demonstrated in Fig. 6. In order to obtain ASIL of the hardware design, it is necessary to calculate hardware architectural metrics, which their definitions and equations are provided in the subsequent sub-chapters. In general, this calculation is performed by failure mode classification, estimation of hardware elements' failure rates, and deployed safety mechanisms' DC evaluation. Finally, using the obtained results, the ASIL of the target design can be determined using pre-defined values in ISO 26262.

In this sub-chapter, initially, failure modes classification is explained in detail. Then, hardware architectural metrics are introduced together with failure rate estimation of hardware components and diagnostic coverage evaluation of safety mechanisms.

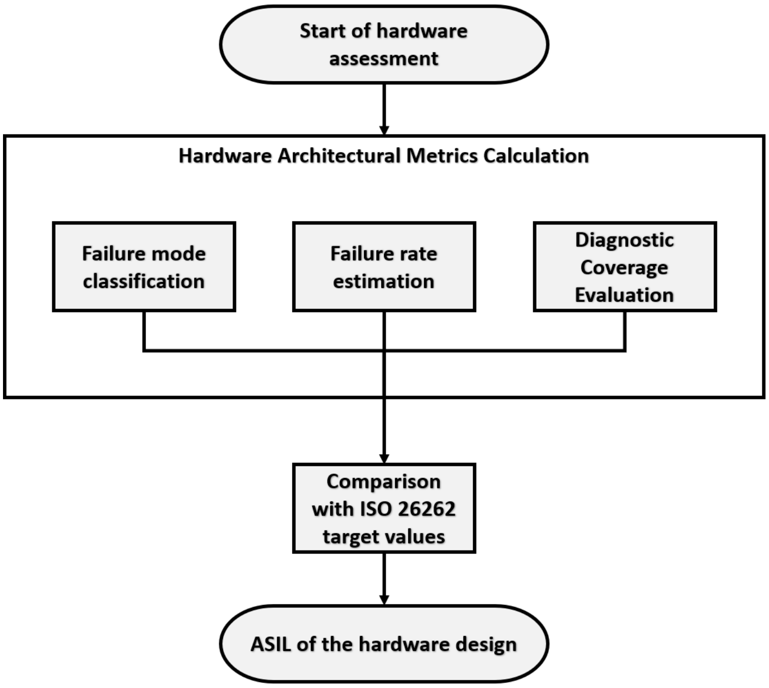

Start of hardware assessment
Hardware Architectural Metrics Calculation
Failure mode classification
Failure rate estimation
Diagnostic Coverage Evaluation
Comparison with ISO 26262 target values
ASIL of the hardware design

Figure 6: Hardware Architecture Assessment Flow in the context of ISO 26262 (adapted from [25])

2.5.1.1 Failure Modes Classification

ISO 26262: Part 5 classifies the failure modes of a hardware element in order to calculate hardware architectural metrics, which will be explained later. Fig. 7 demonstrates the flow diagram of failure mode classification. A failure mode could be classified into the following types, where λ stands for the failure rate [12], which is defined as the frequency in which a component fails:

- **Safe Fault** $(\lambda_{\mathbf{S}})$: A safe fault does not violate a safety goal.
- **Multiple Point Fault** $(\lambda_{\mathbf{MPF}})$: A MPF may lead to a violation of a safety goal in conjunction with another independent faults. There are three types of an MPF;
  - A Perceived MPF $(\lambda_{\text{MPF,P}})$ cannot directly violate a safety goal, but their presence can be perceived by the driver due to performance decline or any other similar negative implication. However, they are not identified by safety mechanisms.
  - A Detected MPF $(\lambda_{\text{MPF,D}})$ is identified by safety mechanism as opposed to a perceived MPF. They are detected within a specified time.
  - A Latent MPF$(\lambda_{\text{MPF,L}})$ is neither identified by a safety mechanism nor a driver.
- **Single Point Fault** $(\lambda_{\mathbf{SPF}})$: It is a non tolerated fault that directly violates a safety goal.
- **Residual Fault** $(\lambda_{\mathbf{RF}})$: A RF is not detected by any safety mechanism and lead to a violation of a safety goal.

Consequently, the failure rate of each safety-related hardware element is stated using (1). Also, we here note that Chapter 2.5.1.3 explains how λ is obtained in the context of ISO 26262.

$$\lambda = \lambda_{\text{S}} + \lambda_{\text{SPF}} + \lambda_{\text{RF}} + \lambda_{\text{MPF}} \tag{1}$$

2.5.1.2 Hardware Architectural Metrics

The effectiveness of safety mechanisms that detect the failure mode needs to be measured by the three metrics listed and detailed below to control faults. In other words, these metrics assess functional safety for hardware components in the context of ISO 26262. Additionally, they support design evaluation and determination of whether or not the deployed safety mechanisms have the sufficient capability to control faults.

These metrics are reported in ISO 26262:Part-5 [12] with their description and formulas. They are defined as follows;

- **Single-Point Fault Metric (SPFM)**: This metric reflects the effectiveness of the safety-related design to protect from a single point and residual faults. Effective safety mechanisms in the design result in few residual faults and higher SPFM. On the contrary, unprotected design parts bring about many single point faults and lower SPFM. Last, many safe faults cause higher SPFM.

  SPFM is calculated using (2), where SR stands for safety-related, λ_{SPF} is the failure rate associated with single point faults, $\lambda_{\text{RF,est}}$ is the estimated failure rates associated with with respect to residual faults, and λ denotes the failure rate corresponding to all hardware faults as calculated in (1). Additionally, (3) shows the calculation of $\lambda_{\text{RF,est}}$, where c is the DC with respect to residual faults.

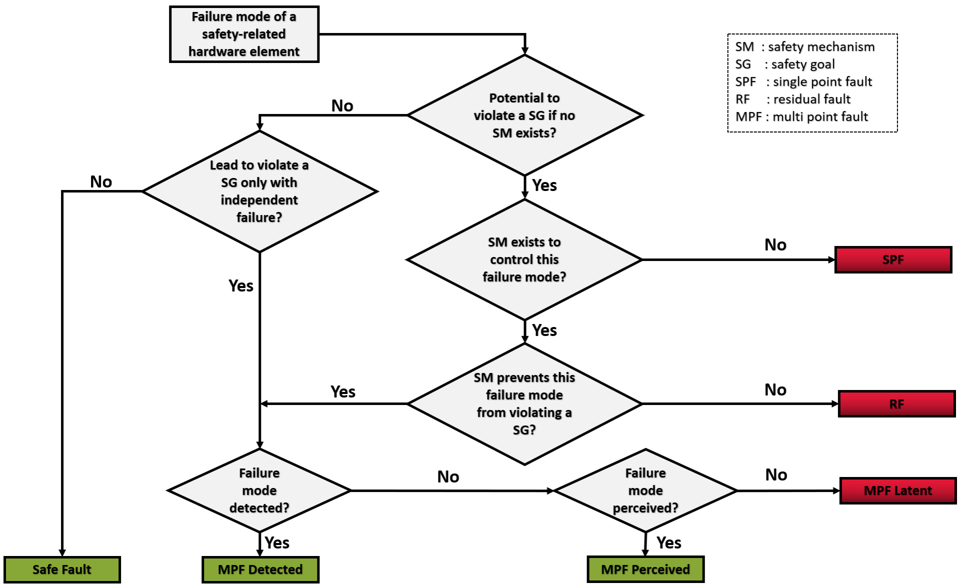

Failure mode of a safety-related hardware element
Potential to violate a SG if no SM exists?
SM : safety mechanism
SG : safety goal
SPF : single point fault
RF : residual fault
MPF : multi point fault
No
Lead to violate a SG only with independent failure?
No
Yes
SM exists to control this failure mode?
No
SPF
Yes
Yes
SM prevents this failure mode from violating a SG?
Yes
No
RF
Failure mode detected?
No
Failure mode perceived?
No
MPF Latent
Yes
Yes
Safe Fault
MPF Detected
MPF Perceived

Figure 7: Failure Mode Classification Flow Diagram [25]

$$SPFM = 1 - \frac{\sum_{SR,HW}(\lambda_{\text{SPF}} + \lambda_{\text{RF,est}})}{\sum_{SR,HW}\lambda} \tag{2}$$

$$\lambda_{\text{RF,est}} = \lambda \times (1 - \frac{c_{\text{DC,RF}}}{100}) \tag{3}$$

- **Latent Fault Metric (LFM)**: This metric reflects the effectiveness of the safety architecture to protect from latent faults, which are a subset of multiple point faults. Many single point faults or residual faults imply higher LFM. Many detected multiple point faults result in higher LFM. Also, many safe faults cause higher LFM.

  LFM is determined by (4), where $\lambda_{\text{MPF,L,est}}$ denotes the estimated failure rate with respect to latent multiple point faults. Also, (5) provides the calculation of $\lambda_{\text{MPF,L,est}}$.

$$LFM = 1 - \frac{\sum_{SR,HW}\lambda_{\text{MPF,L,est}}}{\sum_{SR,HW}(\lambda - \lambda_{\text{SPF}} - \lambda_{\text{RF}})} \tag{4}$$

$$\lambda_{\text{MPF,L,est}} = \lambda \times (1 - \frac{c_{\text{DC,MPF,L}}}{100}) \tag{5}$$

- **Probabilistic Metric of Hardware Failures (PMHF)**: This metric explains that the residual risk of a safety goal violation is sufficiently low [18] [25]. The calculation of PMHF is not provided in ISO 26262, even if it is proposed as one of the alternatives of the probabilistic metrics.

2.5.1.3 Failure Rate Estimation

As shown in Fig. 6, after failures are classified according to the ISO 26262, and as explained in Chapter 2.5.1.1, failure rate estimation can be performed to be used in the metrics introduced above. In another saying, hardware architectural metrics (SPFM and LFM) can be designated by the estimation of failure rates.

For this step, ISO 26262 describes several methods as follows:

- Using recognized industry reliability data books such as IEC 61709, IEC TR 62380, MIL-HDBK-217F.
- Making use of statistical data collected based on tests or field returns.
- Exploiting an expert judgment.

Using one of these methods, the failure rate of a hardware component (e.g., register file, arithmetic logic unit, or instruction memory) is estimated in units of Failure In Time (FIT), which is the number of failures per billion hours. For example, 1 FIT means that the device has a mean time to failure (MTTF) of 1 billion hours [12].

2.5.1.4 Diagnostic Coverage Evaluation

As a next step, DC is assessed. DC shows the effectiveness of the safety mechanism at detecting faults [18], and it is evaluated in this step to calculate the estimated failure rates used in (2) and (4).

DC can be calculated analytically if the target design deploys some standard safety mechanisms such as an ECC. However, even if there is a standard safety mechanism, this computation must be performed considering the specific parts of the logic separately. For example, the DC may be accurate on the data cell of the memory but not for the decoder in front of the memory [18]. In this kind of situation, it might be too challenging to achieve higher safety levels such as ASIL-D. Therefore, it may require to perform more accurate safety analysis techniques such as fault injection. On the other hand, if the target design deploys custom safety mechanisms such as STLs, DC cannot be calculated analytically; hence, functional safety analysis techniques, such as fault injection, must be employed in this case for a more accurate calculation. However, these analysis techniques need optimization to be more efficient and less costly, which is the main scope of this PhD thesis.

2.5.1.5 Comparison with ISO 26262 Target Values

After DC is estimated and evaluated using functional safety analysis techniques, all the prerequisites for the calculation of hardware architectural metrics are ready. Using the formulas given in (2) and (4), metrics are calculated and compared with ISO 26262 target values.

For hardware components, the ASIL requirements regulate the values to achieve for the metrics as shown in Table 1. The calculated SPFM and LFM values are compared to use the values given in this table and finally ASIL of the hardware is determined.

2.5.2 Qualitative Assessment of Hardware Architectures

As opposed to quantitative techniques, qualitative assessment comes into prominence when the target system has shared sources. Therefore, the analysis of dependent failures [26], which is described in Chapter 2.3, is required to investigate the possible common cause and cascading failures between design elements. In other words, it determines that

Table 1: Metrics for Each ASIL

ASIL	Failure Rate	SPFM	LFM
A	<1000 FIT	not relevant	not relevant
B	<100 FIT	$\geq 90\%$	$\geq 60\%$
C	<100 FIT	$\geq 97\%$	$\geq 80\%$
D	<10 FIT	$\geq 99\%$	$\geq 90\%$

safety requirements to reduce the dependencies between the elements are in alignment with safety requirements and have been met.

An example can be given using ADAS in the car. For example, an ASIL-B compliant communication module measures the distance from the front car and sends this data to a cruise controller module that adopts the course of an action in terms of accelerating or braking. However, any glitch or fault on the communication module can create a fatal situation. Therefore, it would be necessary to assess ASIL of this communication module again, which causes additional expenses. In this case, DFA can be used to avoid such a situation.

Additionally, ISO 26262 provides some dependent failure examples such as clock elements, power supply elements, or reset logic, as shared resources exist. Also, typical countermeasures are listed by ISO 26262 as clock monitoring for shared clock resources, physical separation, or isolation for fault avoidance.

2.6 Fault Injection

In Chapter 2.5.1.4, it was stated that fault injection is required in order to have more accurate DC calculation, especially in case the target design deploys custom safety mechanisms. Even if its details are investigated in the subsequent sub-chapters in the context of the research which uses fault injection, in this part, we give the fault injection types and how they differ from each other.

In general, fault injection is the simulation of fault effects on the target design. The purpose is to determine the behavior of faults. Several fault injection techniques have been introduced in academia and the industry. The techniques are classified into the following categories [27] [28] [29]:

- **Hardware-based Fault Injection**: This type of fault injection is performed at the physical level through extra components such as contacts.

  However, using extra hardware components affects the target hardware as it may disturb the power supply by producing voltage and current changes, it may distract the hardware with environmental parameters such as heavy-ion radiation, and it may modify the value of circuit pins.

- **Software-based Fault Injection**: As opposed to Hardware-based Fault Injection, Software-based Fault Injection does not need additional hardware to perform the operation. Instead, it targets the applications and operating systems to reproduce the fault effects caused by faults in hardware. In general, it modifies software execution to inject faults.

  Besides its advantages over Hardware-based Fault Injection, it has some drawbacks. First, it is not possible to inject faults at some places which are not accessible to the software. Second, injection may affect and disturb the workload running on the target system. Third, modeling of permanent faults is too difficult in the software level.

- **Simulation-based Fault Injection**: This type of fault injection is targeted at the early phases of the design implementation, such as RTL or gate-level. Two techniques are widely adopted as *code modification*, and *simulator built-in commands*. The former involves modification of the design description, i.e., VHDL codes, using saboteurs that change the value of a target signal when a fault is being injected. The latter modifies the simulation tools to inject faults and monitor the results on the simulated system. This method supports automation, so the major EDA vendors adopt it.

  The main shortcoming of Simulation-based Fault Injection is the requirement of a large development effort that induces time-consuming campaigns. Also, the accuracy of the design models is another problem that might affect the quality of results.

- **Emulation-based Fault Injection**: Hardware prototypes such as Field-Programmable Gate Array (FPGA) are used to take into account the effects due to the circuit environment. It improves the effectiveness of Simulation-based Fault Injection concerning time and effort overhead.

  However, it has some disadvantages, such as input/output problems or the necessity of high-speed communication links. It also cannot analyze the temporal effects of faults.

3 Functional Safety Verification and Validation Using Fault Analysis Technologies

In this chapter, we explain the Functional Safety Verification and Validation methodology that was presented in **I**, **II**, **III**, and **V**.

3.1 Introduction

Functional safety refers to the absence of unreasonable risk caused by systematic and random hardware failures. Functional safety and especially the analysis of random hardware failures are becoming part of the requirements for designing complex systems. Therefore, tighter integration between functional safety analysis and the standard platform design and verification is required. To achieve the functional safety of SoCs used in automobiles, it is essential to analyze the use cases for all the flow tools according to their probability of introducing errors. This analysis shall evaluate if the malfunctioning tool or its erroneous output can violate a safety requirement. Based on this analysis, ISO 26262:Part 8 [12] covers all aspects of Tool Confidence Level (TCL) and defines critical concepts of confidence and qualification [12]. The TCL assesses the error injection risk of each tool in the flow to document the confidence level for the data processing of each tool. Hence, there is high demand for effective Functional Safety Verification, and Validation methodologies that allow the reduction of costs while maintaining the same levels of safety [5].

In general, there are three complementary technologies used for Functional Safety Verification. First, simulation-based fault injection [18] [30] [31] [25] that shows fault effects if they propagate to considered outputs (safety-critical outputs) and if safety mechanism can detect them. ISO 26262 recommends simulation-based fault injection to achieve safety requirements. However, considering the complexity of modern ICs used in automobiles, the verification environment of simulation-based fault injection campaigns is notably complicated, resulting in long fault injection campaigns. Second, formal analysis can be deployed to assess the fault effects and leverage simulation-based fault injection by identifying safe faults. The advantage of a formal analysis is that it can analyze design behavior considering all test inputs for corner cases [32] [33]. On the other hand, formal analysis and its tools have a major drawback as they cannot analyze all faults in a tolerable time span. This creates a problem in meeting the time-to-market criteria, which is highly demanding considering the number of applications. Therefore, another technology still needs to analyze a large portion of the faults that exist in automotive SoCs. Third, ATPG tools can be employed to decrease the efforts to develop a functional safety verification environment. ATPG tools can generate test benches that are a collection of test patterns to propagate faults. Therefore, simulation-based fault injection can be performed using these test patterns generated by an ATPG tool to achieve better detection rate with less execution time [34] [35]. Nevertheless, ATPG is for manufacturing tests and cannot cover faults that are not in the scan chain structure. Consequently, this chapter proposes a methodology that combines the strength of these three technologies for compliance with ISO 26262 requirements.

The presented methodology aims at verifying the correctness of hardware fault classification of different tools used for functional safety analysis. Moreover, it provides data to support simulation-based fault injection campaigns, the adopted technique by both industry and academia. The presented technique in this chapter has two parallel flows that generate fault classification reports to be verified against each other. The ATPG tool generates test benches and vectors to provide a high propagation rate in the first flow. Then, this generated testbench is used by the simulation-based fault injection tool to verify the design's functional behavior under each fault. In the second flow, a formal analysis tool is deployed to identify safe faults and determine the behavior of faults that ATPG does not cover. Finally, the generated outputs from two parallel flows are verified to identify possible malfunctions. Also, an increase in the fault detection rate is examined when simulation-based fault injection is performed using test benches and test vectors generated by the ATPG. By doing so, tool output's confidence level increased as required by ISO 26262 [12]. In short, the main contributions of this chapter can be listed as follows:

- An automated flow to increase tool confidence level according to ISO 26262.
- Straightforward methodology to avoid extensive tool qualification requirements driven by ISO 26262.
- Combined approach that decreases required functional safety verification effort by identifying safe faults with the help of a formal analysis tool.
- Above 99% fault detection rates on the tested designs supports ISO 26262 Functional Safety Verification, using ATPG test benches in the simulation-based fault injection instead of deploying functional test benches.

In this chapter, we explain functional safety analysis technologies and how they are combined to improve the tool confidence level in the context of ISO 26262. Also, use of ATPG test vectors and test benches to increase detection rates is explained with the experimental results.

3.2 Related Works

Besides the works listed in the Introduction, some other works investigate the use of fault analysis technologies. In this sub-chapter, we explain some of them to show the relevance of the presented chapter.

[36] depicts the challenges of tool qualification in the context of ISO 26262. The authors provide semi-automatic qualification of verification tools, using a monitor and fault injection, to reduce and minimize qualification process costs. They highlight the importance of tool modification, especially in the presence of modifications. The authors also underline the importance of automation to reduce the effort required for tool qualification. Moreover, [37] deploys formal analysis and fault injection to determine the effect of faults in a design. The authors' approach is to develop a fault injection model that allows the use of formal analysis, which is symbolic simulation. They also emphasize the importance of automation and avoiding complex formal property definitions. Furthermore, [38] combines simulation and formal analysis to speed up fault injection campaigns. They identify non-achievable states of faults using formal analysis and reduce the fault injection time by eliminating such faults. Additionally, [39] and [40] employ combination of formal analysis and fault injection. These works use formal algorithms to derive some results that reduce the fault injection workload. They also correlate the obtained results with the ISO 26262 functional safety metrics. Finally, [18] ties functional safety analysis to the traditional EDA flow.

Besides the works listed above, this chapter presents a methodology that incorporates three technologies: simulation-based fault injection, formal analysis, and ATPG. The main difference of this chapter is the use of these three tools together to verify the tool outputs and increase fault detection rates using ATPG test benches and test vectors in simulation-based fault injection.

3.3 Functional Safety Analysis Technologies

This sub-chapter details how functional safety analysis is implemented by simulation-based fault injection, formal analysis, and ATPG. Each technology is explained in terms of its strengths and weaknesses.

3.3.1 Simulation-based Fault Injection

Simulation-based fault injection is a well-known technique used to quantitatively assess the design's ability to cope with random hardware failures. It is available in a variety of tools that are able to perform fault injection in different abstraction levels such as RTL or gate-level. The flow of simulation-based of fault injection is demonstrated in Fig. 8 and detailed below:

- Input Definition and Elaboration: At this step, inputs are given to the tool. These are Testbench, Design, and Fault File Specification. The fault specification file is a text file that specifies a list of modules, instances, or signals as targets for fault instrumentation. It can also specify sub-hierarchies or signals from this list to exclude from the selected targets. Furthermore, one or more fault types (SA0, SA1, SET, or SEU) for each node can be determined. Finally, logical collapsing is applied (if the design is gate-level), an optimized fault list (that includes prime faults) is generated, and the tool instruments faults into a design before the Good Run.
- Good Run: This is the fault-free simulation run to generate reference (good) values for the fault injection. Observation points are set in this phase for comparing the faults between Good Run and Fault Run (the next phase).
- Fault Run: This step is the simulation run to inject faults into the design. The goal is to verify that a fault will be observed at some specific point in the circuit. The number of simulations can be any number up to the number of nodes in the fault list specified in the Elaboration phase. For each Fault Run, the observation points are compared against the reference values from the Good Run.
- Fault Campaign Results: In this step, all fault runs are merged in a single, cumulative report. Each fault is shown as Detected or Undetected. Faults that do not produce changes in the observation points are classified as Undetected. If a fault creates a change in the observation points, it is classified as Detected.

Some advantages of simulation-based fault injection are listed below [27]:

- It supports all abstraction levels such as electrical, RTL or gate-level.
- It demands low-cost automation as it does not require special-purpose hardware.
- It has full control of both fault models and injection mechanisms.
- It can easily be integrated into existing design flows because it can be performed using the same software application that runs on the field.
- It is able to model both permanent and transient faults.

Drawbacks of simulation-based fault injection can be summarized as follows [27] [41]:

- It is time-consuming due to the length of the experiments. In other words, it requires the simulation of fault-free design (Good Run) as well as simulations of the design in the presence of the enormous number of faults.

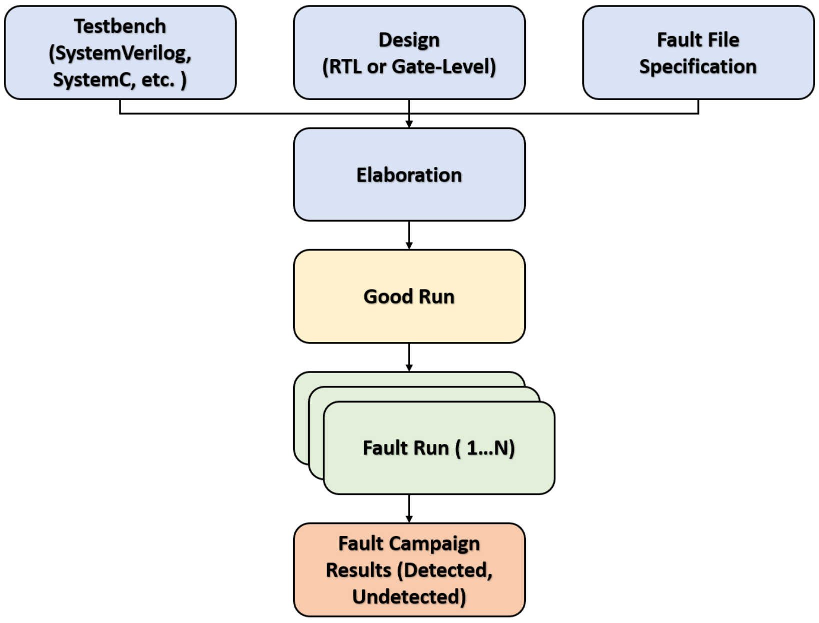

Testbench (SystemVerilog, SystemC, etc.)
Design (RTL or Gate-Level)
Fault File Specification
Elaboration
Good Run
Fault Run (1...N)
Fault Campaign Results (Detected, Undetected)

Figure 8: Simulation-based Fault Injection Flow

- It may have incomplete results. For example, when there are undetected faults as a result of campaigns, this is considered a weak result of the simulation [5] due to the fact that a different test stimulus may cause fault propagation, i.e., different results.

Additionally, if there are no test stimuli that propagate the fault to the observation points, an analysis must prove this. For that reason, it is necessary to develop complex test benches and additional techniques for the analysis of undetected faults. In this case, formal analysis comes to the fore, as explained in the next sub-chapter.

3.3.2 Formal Analysis

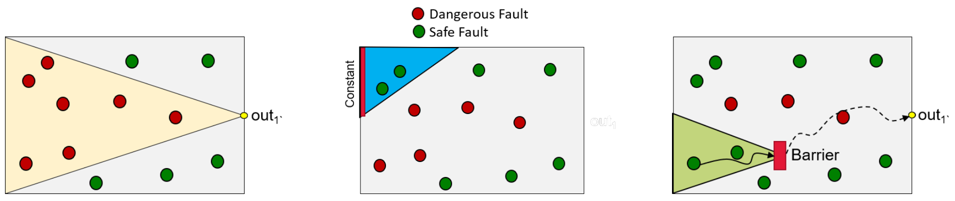
Safe fault identification requires proof that any test stimulus cannot test a fault. Formal analysis can be used for this purpose because it has a global context that is not limited to a specific time or state. In other words, the formal analysis considers all possible test stimuli. Therefore, formal analysis can exhaustively prove that a fault can never produce any failure in the observation points. Thus, these types of faults can be considered safe and do not require further fault injection. By doing this, the faults to be injected in simulation-based fault injection can be reduced so the fault injection campaigns can be optimized in terms of the required time.

Different EDA vendors have their own formal analysis approaches; however, all of them apply a similar methodology for analyzing faults. Basically, the tools perform two analysis techniques: Standard Formal Analysis and Advanced Formal Analysis. The Standard Formal Analysis is also known as structural fault analysis, and the testability of the faults is determined by verifying the following three analyses:

- Out-of-COI: If the fault does not have a physical connection to the observation points, it is determined as safe. Fig. 9a illustrates this. As shown, the faults in the COI of out_1 are dangerous, whereas the others are classified as safe.

- Activation: If the fault is injected on a node that is a constant 0 or 1, then it is safe. Fig. 9b illustrates that faults on the constant signals are safe.

- Propagation: If the fault cannot be observed in one of the observation points due to any barrier, it is safe. Fig. 9c explains that some faults cannot propagate to out_1 because of the barrier, so they are safe.



(a) Out-of-COI Analysis (b) $y = 3sinx$ (c) $y = 5/x$

Figure 9: Standard Formal Analysis Techniques.

A fault that does not pass the above-listed verification steps is classified as safe by the formal tool. Considering Advanced Formal Analysis, which is illustrated in Fig. 10, the formal tools assess functional activation and propagation of the faults. Activation of the faults is to check if the fault can be functionally activated from the inputs. Moreover, propagation of the faults is to check if the fault can propagate to the functional observation points. As shown in Fig. 10, formal properties are automatically generated by the formal analysis tool to verify the fault propagation effects. Then, all possible input stimuli are used to validate fault propagation to the observation points. Finally, results are compared between Bad Machine where the faults are injected and Good Machine, similar to simulation-based fault injection.

As opposed to Standard Formal Analysis that classify faults as only safe or unknown, Advanced Formal Analysis uses the following three classifications for each fault:

- Safe: Faults that cannot be activated or propagated to the observation points.

- Dangerous: There is a combination of test inputs that propagates a fault to the observation points. In this case, the fault is dangerous.

- Unknown: All the faults which are not safe or dangerous.

As mentioned above, formal tools automatically generate formal properties (manual formal property generation also is possible) to perform Advance Formal Analysis. Then, these properties are verified with respect to all possible input stimuli. However, this verification process is time-consuming and expensive. Also, considering today's complex designs in some areas, such as automotive, it is not possible to evaluate all possible test in-puts, so the formal tools cannot classify all faults in the design. Therefore, formal analysis is often applied as a last resource on the faults that were not classified after simulation-based fault injection [5].

EDA vendors integrated fault injection and formal analysis to reduce the effort and complement different strengths of simulation-based fault injection and formal methods. This integration allows the deployment of the Standard Formal Analysis before the simulation-based fault injection. Hence, the number of faults to be analyzed can be reduced by leveraging results for safe faults. Furthermore, Advanced Formal Analysis can be run only on

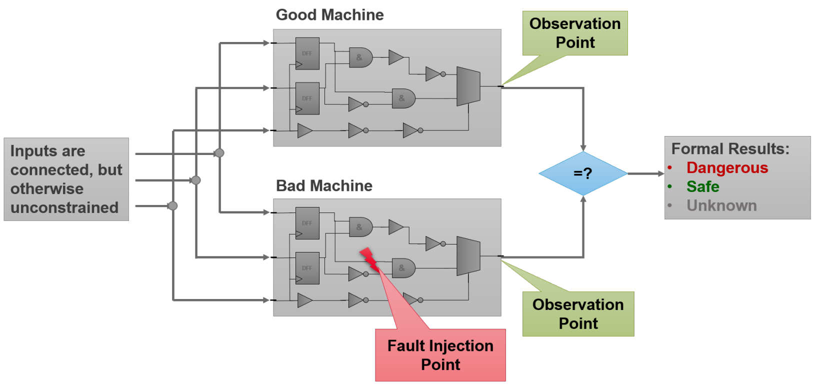


Figure 10: Advanced Formal Analysis

the remaining undetected faults after performing simulation-based fault injection, making fault analysis more efficient.

Even with this integration mentioned above, the development of the test environments is still challenging and time-consuming. Moreover, Advanced Formal Analysis of formal tools can still not classify all the faults in the complex designs. In this context, ATPG appears as a possible alternative solution to generate test stimulus that can be used for the simulation-based fault injection.

3.3.3 Automatic Test Pattern Generator

Any kind of testing is always a matter of controlling the device that is being tested and observing its behavior. In the case of an IC, controlling the chip through the input pins and observing it through the output pins is the only way. This is because accessing internal nodes to control or observe their states is not possible.

There are millions of faults to be analyzed for ICs used in complex areas such as automotive or space applications. Therefore, the amount of computation can be overwhelming. The industry has adopted many simplifications to reduce the test time in manufacturing. One of them is to use functional simulation patterns. However, this is not sufficient as it is impossible to have a complete test using functional patterns. Additionally, as ICs get larger and complex, the observability and controllability of logic states within the chip demand more effort to create the functional patterns. Thus, structural testing is developed in order to achieve a nearly complete test of a chip. It refers to testing the circuit gate-by-gate and net-by-net to ensure that each gate works and that all the interconnections are intact and correct. Also, it does not depend on the functional knowledge of the IC. Tests can be generated automatically, and all forms of APTG use structural test algorithms. In short, ATPG verifies that the IC was built as designed, so it does not verify that the design performs the intended function.

In general, ATPG tools receive a gate-level description of an IC and specification of the scan chains as inputs. Then, it verifies whether the implemented scan chains ensure the required level of testability or not. If affirmative, it generates a fault model and test patterns to assure propagation of fault effects to the design outputs [5]. If unsuccessful, the fault is marked as untestable, and another fault is selected.

The test patterns and expected outputs are programmed in Automated Test Equipment (ATE) to be used in IC manufacturing tests. ATE applies the test patterns in the inputs of the circuit, as explained above, and monitors the observation points (design outputs) to verify if the obtained values are the expected ones. In this chapter, a similar approach is applied using simulation-based fault injection. The test patterns generated by the ATPG tool are utilized by a simulation-based fault injection tool, and observation points are monitored. During the Good Run, the simulator records the good or expected values of the observation points. Then, during the Fault Run, each fault is simulated using the same inputs and monitoring the observation points. By doing this, it is possible to use the propagation capabilities of ATPG to identify behavioral changes caused by injected faults [5].

Besides its fault propagation potential as the main strength, ATPG has some weaknesses. First, it focuses on manufacturing tests, and the estimated results must be demonstrated through simulations. Also, faults out of the scan chain cannot be considered by ATPG.

Table 2 summarizes strengths and weaknesses of each technology presented above. This summary proves that there is a high need for a methodology that combines the strengths of these three technologies to overcome their weaknesses. The next sub-chapter presents a Functional Safety Verification and Validation methodology using these three technology.

Table 2: Comparison of Fault Analysis Technologies [5]

Technology	Strengths	Weaknesses
Simulation-based Fault Injection	- Comprehensive behavior analysis - Recommended by ISO 26262	- Single test input at a time - Too many simulations to propagate all faults - High testbench development efforts
Formal Analysis	- Global context: analysis of all possible test inputs - Analysis of untestable faults	- Time-consuming as it has a global context - Cannot determine behaviour of all faults
ATPG	- Automatic generation of test patterns - High fault propagation rate	- Focus on manufacturing test - Cannot reach corner cases

3.4 The Proposed Methodology

This sub-chapter describes the developed application that combines three fault analysis technologies as an efficient methodology for ISO 26262 Functional Safety Verification. The presented methodology emphasizes the strengths of Simulation-based Fault Injection, Formal Analysis, and ATPG to generate a comprehensive fault analysis report at the end of the flow.

Fig. 11 demonstrates the developed application named Fault Checker [5]. This application aims at automating the execution of the three technologies. It implements a generic and configurable control flow. The application presented in this chapter deploys Cadence tools; however, it is also applicable to other tool vendors. At the end of the flow, reports are generated and saved in a common format that identifies tool malfunctions and detailed analysis of faults behavior.

The Fault Checker needs user inputs to start and control the execution of each tool. These are design files such as libraries and a netlist of the target design. Moreover, the user must provide fault targets and observation points to be used in simulation-based fault injection flow. Finally, for ATPG flow, it is necessary to specify scan chain pins description, which defines design pins associated with the scan chain. After defining all user inputs, the Fault Checker can start executing the Formal and ATPG flow in parallel (using different CPUs) as these flows are independent of each other. On the other hand, Simulator Flow must receive ATPG Test Bench and Test Vectors from ATPG flow to start the execution. Therefore, after ATPG flow is completed, Simulator Flow starts fault injection

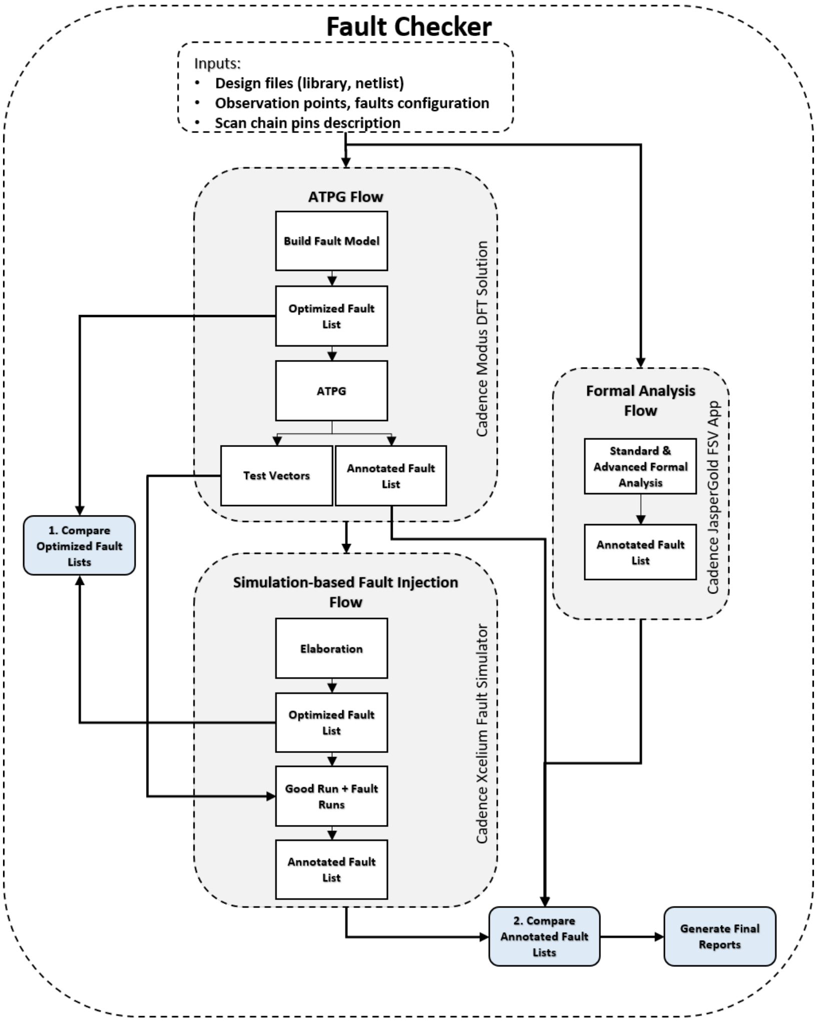

Fault Checker
Inputs:
• Design files (library, netlist)
• Observation points, faults configuration
• Scan chain pins description
ATPG Flow
Build Fault Model
Optimized Fault List
ATPG
Test Vectors
Annotated Fault List
Cadence Modus DFT Solution
Formal Analysis Flow
Standard & Advanced Formal Analysis
Annotated Fault List
Cadence JasperGold FSV App
1. Compare Optimized Fault Lists
Simulation-based Fault Injection Flow
Elaboration
Optimized Fault List
Good Run + Fault Runs
Annotated Fault List
Cadence Xcelium Fault Simulator
2. Compare Annotated Fault Lists
Generate Final Reports

Figure 11: Fault Checker Execution Flow [5]

using outputs of ATPG flow as workload. At the end of each flow shown in Fig. 11, the reports are generated by the tools and parsed to a common format. Finally, at the end of Fault Checker flow, the relevant data is fetched and compared based on rules that associate the classifications used by each tool [5] [3]. If a rule is not obeyed, a Warning tag is written in the reports to inform the designer about a fault that requires further analysis.

The Fault Checker application applies two fault list comparisons as follows.

- The first is to compare optimized fault lists generated by ATPG and simulation-based fault injection tools. This is performed to check if the simulation-based fault injection tool's fault optimization achieves the same capabilities as the ATPG tool, which is accepted as a reference due to its usage for long years in the industry. In other words, the purpose is to prove that simulation-based fault injection tool contains all instrumentation and optimization potential. This step checks whether the same faults are instrumented, whether they have the same number of prime faults (after logical collapsing), and whether they have the same number of safe faults.

- The second is to compare annotated fault lists generated by all the tools deployed in this chapter. Table 3 provides an example of reports generated by the Fault Checker application. This report includes Signal Name, Fault Type, Classification results from three technologies as shown in Fig. 11, and Fault Checker Result. For example, the signal "dut.u0.sig2" in Table 3 is classified as Undetected by the simulation-based fault injection tool and Ignored by the ATPG tool. Nevertheless, the Fault Checker assigns a Warning tag to this row due to the fact that formal analysis identified at least one test stimulus that can propagate SA0 fault on "dut.u0.sig2" to an observation point. This information can be used on a next fault injection campaign to accomplish detection of this fault. The next example in Table 3 is the signal "dut.u0.sig1". SA1 fault on this signal is classified as Safe by the formal analysis tool, whereas the other tools classified this fault as Undetected and Ignored, respectively. Safe classification of formal analysis tools proves that no test stimulus propagates SA1 fault on "dut.u0.sig1" to any observation point, i.e., the fault is Untestable. Therefore, this information can be used as a contribution to achieving ISO 26262 metrics.

3.5 Experimental Setup and Results

In this sub-chapter, we describe the validation process of the proposed technique. In the beginning, we detail the experimental setup, the tool configurations, and the designs used to verify the proposed technique. Then, results are provided and described the benefits and limitations of the proposed solution. Two approaches are adopted as validation aspects. First is the detection of malfunction in the three tools using the detailed report as explained in the previous sub-chapter (comparison of optimized and annotated fault lists). Second is the application of fault analysis results to support functional safety verification of the target designs (when simulation-based fault injection tool deploys test benches and test vectors generated by ATPG tool).

3.5.1 Experimental Setup

As shown in Fig. 11, there are three tools deployed for the proposed methodology. This work adopts Cadence® JasperGold Formal Verification Platform Functional Safety Verification, Cadence Xcelium™ Fault Simulator (XFS), and Cadence® Modus DFT Software Solution ATPG component. However, the proposed methodology remains applicable to other tool flows as well.

Regarding the example target designs used in this chapter, different levels of complexity and the availability of functional test benches are considered. Complexity is about the number of faults in the design. When it comes to ISO 26262, all cell ports in the gate-level representation of a design should be analyzed for faults. The selected designs were synthesized using the standard cell reference libraries provided with Cadence 45nm Generic

Table 3: Fault Checker Report Example [5] [1]

Signal Name	Fault Type	Formal Analysis Classification	Simulation-based Fault Injection Classification	ATPG Classification	Fault Checker Result
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

Process Design Kit (GPDK) to obtain a gate-level netlist. These designs are listed as follows and available in [42]:

- Up-Down Counter: This is 4 bits adder design that contains 81 cell ports
- Memories: This design has two memories with Cyclic Redundancy Check (CRC), containing 1391 cell ports
- AC97: This is an Audio Codec Controller compatible with Wishbone bus, containing 28610 cell ports
- Conmax: This design is an interconnect matrix IP core with a parameterized priority-based arbiter containing 76727 cell ports.

3.5.2 Experimental Results

In order to check fault instrumentation capability of simulation-based fault injection tool as shown in Fig. 11, optimized fault lists for Up-Down Counter and AC97 designs are compared in the beginning. As a result of this analysis, there is no difference identified in optimized fault lists' of Up-Down Counter. However, 12 faults were different in AC97. The reason for this is the different collapsing approaches of ATPG and simulation-based fault injection tools. In other words, collapsing methods of these tools affect the results. For example, the ATPG tool does not collapse faults on primary inputs; however, these faults are collapsed by the simulation-based fault injection tool. Nevertheless, this behavior does not change the functionality of the circuits [1].

Then, Table 4 provides the Detection Rate when generated test benches and test vectors by ATPG is deployed in simulation-based fault injection. This table also shows the comparison of annotated fault lists. This is shown as the number of PASS and WARNING indications identified by Fault Checker. Below we explain the Fault Checker results as demonstrated in Table 4:

Table 4: Fault Checker Results [5]

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

- Up-Down Counter: As this is the simplest design among selected ones in this chapter, the three fault analysis technologies determine that all faults in Up-Down Counter can propagate to observation points, which means that Detection Rate is 100%. Moreover, the Fault Checker validated that all faults have equivalent classifications, so there is no WARNING tag in the final reports.
- Memories: The Fault Checker identifies 6 faults with different classifications for this design, so 6 WARNING tags are included in the final reports. These WARNINGs are about faults that are classified as Safe by the Formal Analysis and undetected by the Simulation-based Fault Injection. In other words, the Formal Analysis tool proves that these 6 faults are safe and can be excluded, improving results for ISO 26262 metrics calculation.

- AC97: 118 faults are identified with discrepant classification as shown in Table 4. Out of this 118,
  - 49 faults were classified as safe by formal analysis and undetected by simulation-based fault injection; hence, they can be declared as safe.
  - 23 faults were classified as dangerous by formal analysis and undetected by simulation-based fault injection. This shows that these faults can be detected in simulation-based fault injection by applying the results from formal analysis as test inputs.
  - 46 faults were classified as undetected by simulation-based fault injection and ATPG. Formal analysis cannot identify them, so they are classified as unknown by formal analysis tool. This indicates that tools could not define the possible behavior of these 4 faults; hence, a manual analysis of these faults is required.
  - 6 faults were in cell ports connected to the power or ground. These faults are not relevant for ISO 26262 functional safety verification.
- Conmax: 263 different classifications between tools were detected by the Fault Checker.
  - 7 faults were dangerous according to the formal analysis and undetected by simulation-based fault injection. This means that results from formal analysis can be applied to simulation-based fault injection to detect these faults; hence, detection rate can be increased.
  - 256 faults were classified as unknown by formal analysis, undetected by simulation-based fault injection, and redundant by ATPG. The designer must manually analyze these faults as the classifications are not precise.

Moreover, in this chapter, the results of simulation-based fault injection when a functional testbench is deployed and when the Fault Checker runs fault injection campaigns with test vectors generated by ATPG, as demonstrated before in Fig. 11. Table 5 shows the results of this comparison. As a result of simulation-based fault injection using Functional Testbench, 71,50% and 81,66% coverage is obtained for AC97 and Conmax, respectively. On the other hand, when the Fault Checker is deployed (so simulation-based fault injection is performed with ATPG test vectors), the numbers are increased to 99,77% and 99,80% for AC97 and Conmax, respectively. This shows that APTG test vectors have higher fault propagation strength compared to Functional Testbenches. Increasing the detection coverage is quite important because undetected faults after simulation-based fault injection must be analyzed. Experts usually do this analysis manually; however, this is time-consuming, error-prone, and expensive. Also, it would be necessary to develop new tests and repeat fault injection campaigns to reduce the number of undetected faults. However, this increases the development time of ICs and makes it difficult to meet time-to-market criteria. Thus, it is useful to benefit from ATPG test vectors, which have higher propagation strength, as done in the Fault Checker.

3.6 Chapter Conclusions

In this chapter, the combination of three fault analysis technologies is introduced. The proposed methodology provides for a high degree of tool confidence level to detect tool errors in the context of ISO 26262. First, each fault in the target design is classified by three tools and compared against each other. Then, the Fault Checker identifies if there is

Table 5: Fault Detection Comparison [5]

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%

a discrepancy between these classifications. Moreover, the proposed methodology makes compliance to ISO 26262 easier with the identification of Safe faults. In other words, the identification of safe faults decreases the total number of faults to be analyzed in Simulation-based fault injection and increases ISO 26262 metrics such as Diagnostic Coverage [12]. Additionally, the use of ATPG test benches and test vectors in Simulation-based fault injection increases the detection rates as ATPG has more strength to propagate the faults to the observation points.

In summary, the proposed methodology provides an efficient and automated way for Tool Qualification and error detection in the tool outputs. Moreover, safe fault identification decreases functional safety analysis as they can be excluded from fault injection campaigns. This also increases metrics guided by ISO 26262. Furthermore, supplementary data provided by the Fault Checker can be utilized to support other fault injection campaigns.

4 The AutoSoC Benchmark Suite

In this chapter, we explain the Automotive SoC (AutoSoC) Benchmark Suite that was conceptualized in **VII** and used as a case-study in **X**.

4.1 Introduction

Especially with the development of autonomous vehicles, functional safety analysis is becoming more challenging. Also, functional safety analysis in the context of ISO 26262 requires representative automotive SoCs to perform and validate developed methods and techniques. However, there are not enough hardware and software resources accessible to the researcher and industry to assess if their results have the quality. Therefore, there is a high need for open-source and representative SoCs that enable comprehensive functional safety verification.

Considering existing commercial solutions, in general, all of them share some similarities in terms of architecture. For example, the availability of multiple CPUs is a common characteristic of SoCs used in automobiles. In other words, running safety-critical applications and other applications in separate CPUs is a common practice. Regarding safety mechanisms, DCLS is the most commonly deployed type. Concerning memories, RAMs, and caches, the industry mainly utilized ECCs and Parity, also highly recommended by ISO 26262. Furthermore, communication protocols such as CAN, SPI, Ethernet are presented in automotive SoCs. In addition, some application-specific hardware components such as Graphics Processing Unit (GPUs) or image processing units are also provided targeting ADAS. Additionally, security components become more of an issue in automotive applications, so peripherals implementing cryptography algorithms are included in SoCs. Finally, general communication protocols like Universal Asynchronous Receiver-Transmitter (UART), Joint Test Action Group (JTAG) are available in all commercial solutions.

The AutoSoC is an open-source benchmark suite formulated based on the existing commercial solutions and adopted industry development techniques. It incorporates all required elements in the format of a configurable SoC, such as hardware models, operating systems, and software applications. It is developed to support research in the automotive domain by providing varied hardware configurations, safety mechanisms, and representative software applications, fulfilling the requirements driven by ISO 26262. Moreover, the selected SoC architecture is available at RTL and gate-level to comply with industry demand. Also, it has safety mechanisms that enable extensive functional safety analysis thanks to its availability in multiple configurations. Concerning software resources provided with the AutoSoC, compilers, debuggers, operating systems, and software test libraries are presented and developed to support varied analysis. Automotive Cruise Control is also included as a software application to demonstrate representative use cases. This application runs on RTEMS operating system or as bare metal, so it is conceivable to investigate the fault effects on these different platforms. Last, the AutoSoC incorporates communication protocols such as CAN and UART as in commercial solutions.

In summary, the AutoSoC is a promising initiative for an open-source SoC benchmark suite for researchers who wants to work on automotive applications. This chapter introduces the general architecture of the AutoSoC, its peripherals (UART and CAN), and available software applications. Also, preliminary functional safety analysis performed in RTL and gate-level is shown.

4.2 Related Works

This sub-chapter lists some of the related works in functional safety targeting automotive ICs. [43] and [44] explore the hardware fault-tolerant architectures such as Dual Modular Redundancy (DMR) and DCLS. On the other hand, [45] examines diverse compiling software fault-tolerant architecture and analyzes the effects of faults in the context of safety-critical software systems. Similar fault tolerance architecture investigation is also available in operating systems. [46] analyses the efficiency of traditional fault tolerance methods on parallel systems running the Linux operating system. Furthermore, functional safety compliance to ISO 26262 is researched in [25] that highlights simulation-based fault injection as a key step in order to be in compliance with ISO 26262. In addition to this, there are several works such as [31] and [47] that focus on fault injection optimization techniques to speed-up campaigns and reduce the time cost of the functional safety analysis process. Moreover, cross-layer functional safety analysis approaches, which are the use of different abstraction levels, are also presented in [48] and [49]. These works perform fault-effect analysis on virtual prototypes of automotive SoCs, showing the advantage of using higher abstraction levels instead of RTL or gate-level.

Although all the works listed above contributed to the state-of-the-art functional safety analysis, they do not perform their analysis on representative automotive SoCs. Additionally, not all cases are executed with diverse software applications that can also run on operating systems. Finally, as these works are not open-source to the community, other researchers cannot compare the results' quality. The AutoSoC benchmark suite solves these problems by presenting an open-source and comprehensive automotive SoC package with its hardware and software resources.

4.3 General Architecture of the AutoSoC

Fig. 12 demonstrates the architecture of the AutoSoC. First of all, functional blocks are defined based on the characterization of industrial solutions. These blocks provide the minimum set of requirements for an automotive representative SoC. Moreover, all the defined functional blocks explained below are modular, meaning that different SoC versions can be implemented using selected blocks. Concerning the functional blocks shown in Fig. 12;

- Safety Island performs all safety-critical processes, as a similar block is available in all commercial SoCs. It includes a CPU and memories, and both of them are covered by safety mechanisms to comply with ISO 26262.

- Application-Specific Block is to process all the other applications such as video or image processing. Like Safety Island, it includes a CPU and SW stack (memories), but Application-Specific Block does not need to be protected by safety mechanisms.

- Interconnect Block implements the Wishbone Bus, which is the hardware computer bus structure of the AutoSoC. This block is responsible for internal SoC communication.

- Infrastructure Block has general-purpose communication protocols such as JTAG, ADBG, and UART.

- Automotive Block deploys CAN, the most common in-vehicle communication protocol adopted by the automotive industry.

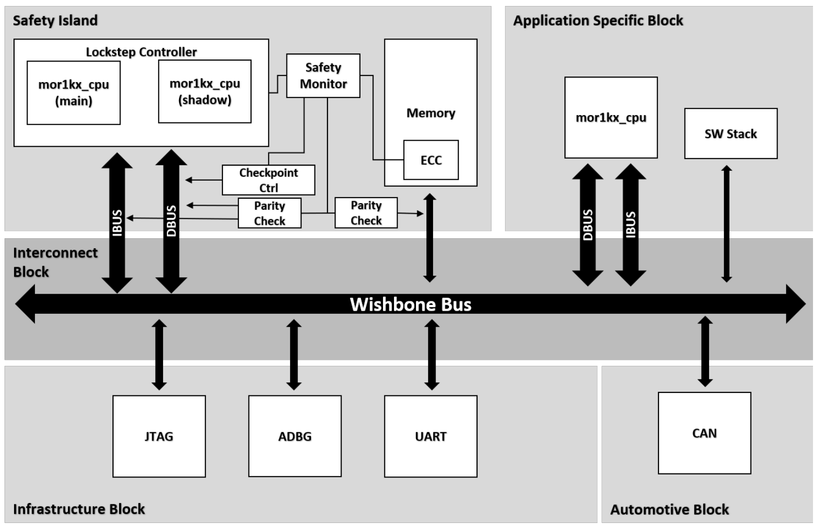


Figure 12: Architecture of the AutoSoC [7]

Fig. 12 shows all the available functional blocks of the AutoSoC. However, the modular structure of the AutoSoC makes adding more blocks possible. For example, security extension can be done to comply with the new security standard ISO 21434 [50]. Security block examples are Advanced Encryption Standard, Data Encryption Standard, or Hash.

In the following sub-chapter, the hardware components of the AutoSoC are investigated.

4.3.1 Hardware Components

The AutoSoC implements mor1kx implementation of openRISC [51] as the main CPU. It is selected as it includes a variety of support tools and resources with the help of an active community. Moreover, mor1kx allows designers to customize the core with its integration capability. It is designed as parametric, making it easy to modify the CPU according to the needs or add new peripherals.

The mor1kx package includes the CPU, memory, UART, JTAG, and a debug unit, all connected via Wishbone bus [7]. Moreover, a testbench, which loads software applications to the memory and provides a connection to the JTAG for debug purposes, is provided in the mor1kx package. By using the existing infrastructure of the mor1kx package, the AutoSoC's basic features were tested quickly, and all the other developed applications were reused in these provided files.

Furthermore, the AutoSoC is prepared at both RTL and gate-level abstraction levels to support more comprehensive research. The synthesis is performed using Cadence GPDK045 (45nm CMOS Generic Process Design Kits).

Concerning the peripherals shown in Fig. 12, the following sub-chapter investigates CAN and UART in more detail as these two peripherals are used in Chapter 6.

4.3.1.1 Controller Area Network

The CAN is a communication bus standard introduced by Bosch in 1986. It is intended to work in the automotive field for serial communication applications among microcontroller units. The CAN has several benefits; it is low-cost, and it has the ability to self-diagnose and repair data errors. These features promote CAN's popularity in automotive and some other industries such as medical or aerospace [52]. As it represents the automotive industry's challenges, it is integrated into the AutoSoC.

The AutoSoC Benchmark Suite has open hardware implementation of the SJA1000 [53] which is a stand-alone controller for the CAN, developed by Philips Semiconductors in the early 2000s. Fig. 13 shows the block diagram of SJA1000 CAN. These are explained below:

- The CAN Transceiver is a module to connect other nodes to the CAN.
- The CAN Core Block controls the reception and transmission of CAN frames.
- The Interface Management Logic implements the CAN interface as a link to the host CPU through its set of registers. It depicts commands from the CPU, conducts addressing of the registers, send interrupts, and provides status information to the host CPU. Also, this block configures the operational mode of CAN whether it works in *BasiCAN* or *PeliCAN* mode.
- The Transmit Buffer stores messages in Extended or Standard Format for transmission over the CAN network. Also, the CAN Core Block reads messages from the Transmit Buffer whenever the Interface Management Logic forces it.
- The Acceptance Filter comes into prominence when receiving a message. It checks whether the message on the bus has to be stored by the CAN or not. This is done by comparing the received identifier with the Acceptance Filter register contents.
- The Receive FIFO stores all received messages accepted by the Acceptance Filter.

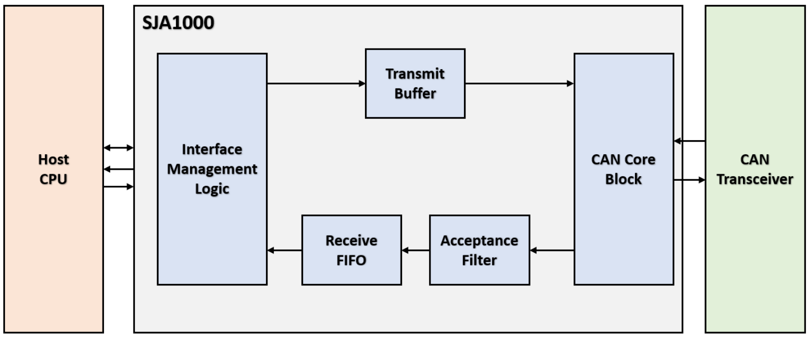


Figure 13: Block Diagram of the Adopted SJA100 CAN [10]

As the AutoSoC Benchmark Suite uses a Wishbone Bus, the adopted CAN is directly connected without the need for bridges between different bus interfaces. When it is required to add another node to be communicated with the Host CPU, the CAN Transceiver provides a straightforward way for connection. Moreover, the AutoSoC Benchmark Suite provides an STL for the self-test of the CAN. As it is explained in [54], the developed STLs implement an effective in-field test for the CAN-based on a functional approach and also provide experimental evidence to demonstrate its effectiveness.

4.3.1.2 Universal Asynchronous Receiver-Transmitter

The AutoSoC benchmark suite includes a UART, which incorporates the industry-standard National Semiconductors' 16550A device features [55]. Furthermore, as it is a well-known and widely-used communication standard by the industry and academia, the AutoSoC benchmark suite covers UART.

UART is a block of circuitry that uses asynchronous serial communication with configurable speed. It operates data transfer by receiving data from a peripheral device or a CPU. Moreover, the UART includes an interrupt system and control capability tailored to minimize software management of the communication link. The UART used in the AutoSoC operates in 32-bit bus mode fully compatible with Wishbone Bus. As depicted in Fig. 14, the UART core consists of Receive Logic, Control, and Status Registers, Modem Control Module, transmit Logic, Baud Generator logic, and Interrupt Logic, as explained below:

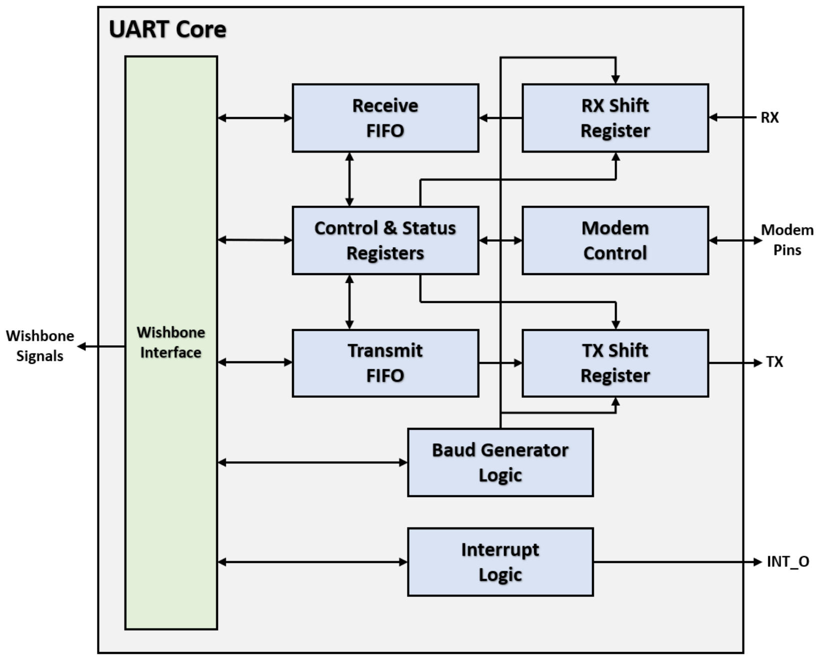


Figure 14: Block Diagram of UART [10]

- Incoming serial messages are received by the RX Shift Register, whose Baud Rate is programmable through Baud Generator Logic.
- Received messages are placed in the Receive FIFO if the incoming messages have no problems.
- TX Shift Register handles the transmission of data written to the Transmit FIFO.
- Control and Status Registers allow the specification and observation of the format of the asynchronous data communication used.
- Modem Control has registers that allow transferring control signals to a modem connected to the UART.

- Baud Generator Logic controls transmit and receive data rates.
- Interrupt Logic allows enabling and disabling interrupt generation by the UART.

The AutoSoC Benchmark Suite includes the above-explained UART and some test programs to experiment with the functionality of the UART to provide a baseline for researchers to develop and validate their approaches.

4.3.2 Safety Components

As shown in Fig. 12, the Safety Island is responsible for safety-critical applications in the AutoSoC. Therefore, it is essential to provide safety mechanisms to persuade that potential faults can be detected and possible harm to the expected functionalities can be avoided [7]. Different safety mechanisms implemented in the AutoSoC are listed below:

- Dual-Core Lock Step with time diversity is deployed as the main safety mechanism in the AutoSoC. Fig. 12 demonstrates that DCLS configuration includes a redundant copy of the main CPU. The main CPU is responsible for controlling the SoC functionality and writing access to the Wishbone bus. Nevertheless, the shadow CPU does not execute write access to the SoC components. It is used to compare the outputs against the main CPU using a Compare Unit, which is not shown in Fig. 12. If the Compare Unit identifies a mismatch between the outputs of main and shadow processors, an alarm is activated for fault detection. DCLS is a widely adopted safety mechanism for the automotive SoCs targeting ASIL D; on the other hand, it increases hardware area due to the necessity of a redundant copy (shadow CPU). Thus, additional safety mechanisms are implemented in the AutoSoC.
- Software Test Libraries (STL) is a software-based safety mechanism used to prevent permanent faults. The semiconductor companies adopt them due to their advantages. For example, STLs can execute the test in the system operating conditions, avoid any overtesting and any area or performance overhead. Hence, STLs are used as a safety mechanism in the AutoSoC benchmark suite as it is a good alternative to DCLS with high hardware overhead.
- ECCs protect internal memories, and they are highly recommended by ISO 26262. Thus, the Safety Island of the AutoSoC includes ECC in its internal memories and RAMs. Also, the external memory, which loads the software applications, is also covered by ECC to avoid propagation of faults to the primary outputs of the Safety Island
- Bus Parity is deployed to shelter the data bus, which is responsible for the data transmission between the memory and the CPU of the AutoSoC. Thanks to the Bus Parity safety mechanism, propagation of faults within the data bus to the CPU or the memory is prevented.
- Checkpoint Control is utilized in case CPUs (when DCLS is active) are stuck in the same software instruction. It checks the data bus to observe predetermined software signatures in specific memory locations.
- Safety Monitor is also included in the AutoSoC. The aim is to integrate all the detection alarms. The Safety Monitor generates an external alarm and an indication of where the fault was detected.

Additionally, the AutoSoC benchmark suite is implemented as a modular SoC, supporting several configurations that enable different combinations of safety components mentioned above. Five different AutoSoC configurations are assembled for different scenarios or use cases. These configurations are summarized in Table 6.

Table 6: AutoSoC Configurations [7]

AutoSoC Configuration	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	+	-	-	+	+	+

4.3.3 Software Resources

The AutoSoC benchmark suite includes several software options. By setting configuration files in the suite, any software application can be run on the AutoSoC. In addition, software applications that run Baremetal and on RTEMS or Linux operating system are provided. Concerning RTEMS, the Automotive Cruise Control software application is developed and used in Chapter 6 for the identification of application-specific safe faults. This application has real-time tasks that read vehicle sensor data, compute actuation, and set engine parameters. Furthermore, several small and middle-size software applications are available, supporting any tests designers want to execute. Moreover, the Automotive Cruise Control software application employs CAN or UART depending on the user to communicate with the external world. This enables functional safety analysis on a complete SoC instead of narrowing the investigation scope to the only CPU.

4.4 Experimental Results

This chapter summarizes the preliminary functional safety analysis performed for the AutoSoC DCLS and AutoSoC STL configurations. Simulation-based fault injection is performed using some selected workloads, and DC is measured, as specified by ISO 26262.

First, fault injection is performed on AutoSoC DCLS when the abstraction level is RTL. Permanent faults (SA0 and SA1) are injected. As shown in Table 7, fault target is defined as only the CPU core, mor1kx_cpu. As a result, 99% diagnostic coverage is measured, meaning that the DCLS safety mechanism can detect 99% of faults in the main CPU (mor1kx_cpu). This number also conforms with ISO 26262 that defines the typical diagnostic coverage for these mechanisms as high. Moreover, with 99% diagnostic coverage, the AutoSoC achieves the ASIL D requirement.

Table 7: AutoSoC DCLS Functional Safety Analysis Results [7]

Fault Target	# of Injected Faults	# of Detected Faults by DCLS	# of Residual Faults	Diagnostic Coverage
mor1kx_cpu	675,504	668,749	6,755	99%

Second, AutoSoC STL configuration is used for the subsequent functional safety analysis. Different than the analysis on AutoSoC DCLS, results are gathered from both RTL and gate-level representation of the AutoSoC to mimic the real use case. The deployed STL programs constitute 16 test programs targeting the CPU (mor1kx_cpu). These STLs make use of three common strategies for software-based-self-test (SBST) paradigm [56] as ATPG-based, deterministic and evolutionary-based [57]. Table 8 shows the analysis results on AutoSoC STL configuration. As it can be seen, fault targets are only Arithmetic Logic Unit (ALU) and Load-Store-Unit. These two instances of the AutoSoC include 42,160 and 60,672 permanent faults in RTL and gate-level, respectively. Table 8 compiles the gathered results showing that the achieved DC on the ALU and LSU in RTL and gate-level. Testable Diagnostic Coverage (TDC) is also reported, considering the redundant and safe faults.

Table 8: AutoSoC STL Functional Safety Analysis Results [7]

Fault Target	**RTL**		**Gate-Level**	
	DC [%]	**TDC [%]**	**DC [%]**	**TDC [%]**
ALU + LSU	68.71	80.04	76.23	85.43

4.5 Chapter Conclusions

With the introduction of autonomous vehicles into our lives more and more every day, functional safety is gaining more importance. However, functional safety, regulated by an ISO standard, is a laborious and challenging process due to the complexity of the ICs and applications used in cars. For this reason, the methods used for functional safety need to be improved and made more efficient. Thus, the demands of the automotive semiconductor industry, which is developing day by day, can be met. In order to do this, researchers need to be able to easily access representative automotive SoCs, where they can test the quality of the results by applying the methods they have developed. However, nowadays, it is not possible to find and work on an open-source automotive SoC.

As a solution to the problems mentioned above, the AutoSoC benchmark suite is proposed in this Chapter. The AutoSoC was developed after a detailed examination of existing commercial products and after determining all the requirements of an automotive SoC. The AutoSoC, which is openRISC based and open-source, includes safety island and application-specific block in accordance with ISO 26262. In addition to this, the Infrastructure Block containing the most used communication protocols such as UART or JTAG and the Automotive Block, including the CAN used for in-vehicle communication, has been added. Furthermore, many safety mechanisms such as DCLS, ECC, and STL have been added to the AutoSoC suite. As these are presented in a configurable form, various functional safety analyses can be performed on the AutoSoC. Also, besides the hardware components just mentioned, the AutoSoC also has an extensive software resource. For example, CCA, one of the most encountered automotive software applications, is provided running on both bare-metal and an operating system such as RTEMS. Finally, the AutoSoC is implemented at both the RTL and gate-level abstraction levels, supporting cross-layer analysis.

In this section, the AutoSoC benchmark suite is explained in detail, and the results of functional safety analysis are shared. According to these results, the AutoSoC stands out as an open-source and comprehensive automotive SoC for all researchers to try its methods.

5 Accelerating Simulation-based Fault Injection Campaigns with Fault List Pruning

In this chapter, we explain techniques to prune the fault list for transient simulation-based fault injection. Following sub-chapters introduce the topic, then present two different techniques as "Dynamic Slicing based Fault List Pruning" **IV**, **VI** and "Mapping based Fault List Pruning" **IX**.

5.1 Introduction

It is necessary to avoid fatal consequences for a reliable IC operation, especially for the systems used in safety or security-critical areas. Therefore, there is a need to analyze possible faults which may occur in these systems. One of the possible faults is transients which are created by an energetic nuclear particle or an electrical source [58]. Moreover, ICs are becoming more susceptible to transient faults because the noise margins have decreased with the use of advanced node technologies, and the sensitivity to alteration in parameter changes such as voltage or temperature has increased [59]. Memories are considered the most vulnerable circuit components to transient faults because they have a high spatial density and store a high amount of data [58]. However, the logic core of an IC has also gained importance recently with the use of advanced nodes [60] [61].

Due to the reasons mentioned above, it is necessary to evaluate the possible effects of transient faults, which pose a significant risk for the reliability of safety or security-critical systems. The way of doing this evaluation is to perform a transient simulation-based fault injection. However, modern ICs contain millions of memory elements and logic components due to their complexity. Additionally, workloads run on ICs execute over trillions of clock cycles, and a transient fault can occur at any execution cycle [62]. As a result, fault space for transient simulation-based fault injection is enormous, making this analysis time-consuming and expensive in terms of the engineering effort. Consequently, developing some techniques to prune the fault list of transient simulation-based fault injection and reduce the execution time is required.

There are two ways to speed-up simulation-based fault injection targeting transient faults as follows.

- First is to *select critical time-steps* at which faults are injected because if a signal is not read at a time-step, an injected fault at this time cannot have any impact on the outputs and becomes undetected. By doing this, the transient fault list is pruned in terms of the time dimension. As shown in Fig. 15, simulation-based fault injection is characterized by several items. The location designates where the fault is built-in. The fault type (model) is either permanent or transient. The injection time defines the start of fault injection. When compared to permanent faults, transient faults must be injected at several time-steps per workload because, as mentioned above, a transient fault can occur at any time. Therefore, critical time-steps need to be identified to speed-up transient simulation-based fault injection.

- Second is to *change the abstraction level*. Fig. 16 shows the design abstraction levels. When simulation-based fault injection is performed in lower levels such as gate-level or circuit level, simulation time, size, and complexity increase. This reflects simulation-based fault injection as an increased execution time. Therefore, it is required to develop a technique that maps faults from lower levels to higher levels.

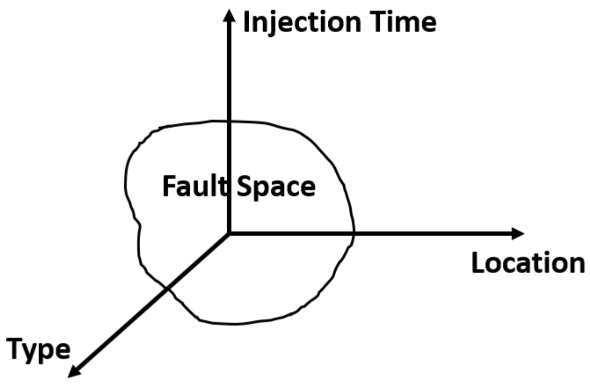


Figure 15: Fault Space Dimensions

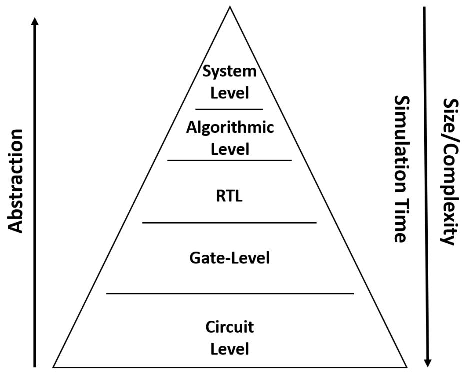


Figure 16: Design Abstraction Levels [63]

Consequently, we developed two techniques for both items presented above. The first one is "Dynamic Slicing based Fault List Pruning" which selects critical faults or time-steps to speed up the simulation-based fault injection. This is detailed in Chapter 5.3. The second one is "Mapping based Fault List Pruning" which transfers simulation-based fault injection from the gate-level to RTL by mapping SET faults to the flip-flops and using the advantage of multiple and simultaneous simulation-based fault injection. This technique is described in Chapter 5.4.

5.2 Related Works

There exist many advanced tools and methods for optimizing simulation-based fault injection. In [64], a tool called VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) is presented that utilizes an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers, which provide the design libraries, to express their knowledge of the fault behavior of their components. Although it provides multi-threaded fault injection and checkpoints and comparison with a golden run to speed up the simulation of faulty runs, the drawback is that it requires modification of the VHDL language itself. [65] proposes MEFISTO-C: A VHDL-based fault injection tool that conducts fault injection experiments using VHDL simulation models. The tool supports a variety of predefined fault models; however, it does not provide specific optimizations to speed up the simulation.

Several approaches to generate the critical fault list to be considered the basis of fault list injection have been proposed. In [66], a method for generating a critical fault list is presented. A data flow graph describes the system under test, the fault tree is constructed by applying the instruction set architecture fault model to the data flow description with a reverse implication technique, the fault injection is performed, and fault collapsing on the fault tree is employed. However, the proposed method is very costly in terms of CPU time, and it is therefore not applicable to systems with high complexity.

[67] presents a new technique and a platform to accelerate and speed-up simulation-based fault injection in VHDL descriptions. Use checkpointing to reload the fault-free state if the design allows starting the fault injection from the clock-cycle of fault injection. In addition, a golden-run fault collapsing technique is utilized that discards all fault injections between read-write and write-write operations of the memory elements. However, the approach does not take advantage of the dynamic slicing benefits. [68] proposes fault collapsing based on extracting high-level decision diagrams from the VHDL model. Although significant speedup can be achieved, the step of efficient decision diagram synthesis from the full synthesizable subset of VHDL remains an issue.

Several papers are dealing with transient fault injection. [69] shows the results collected in a series of fault injection experiments conducted on a commercial processor. Here, the authors inject a fault in a given sequential element at a given instant of time. However, as it is hard to inject a fault in each of the tens of thousands of sequential elements in the processor, the execution is divided into the parts and, for each of these parts, a random fault injection instant is selected. [70] analyses fault injection campaign in the CPU registers by choosing a random instant when the fault is injected. [47] identify the optimal set of flip-flops. However, injection time is randomized uniformly over the active region of the simulation. Similarly, [71] injects a fault randomly in time and location in RT-level. Lastly, [72] deals with single and multiple errors in processors by randomly selecting injecting time and choosing registers. As opposed to these works, our approach shows the fault injection time explicitly instead of random instants.

Dynamic slicing technique is used in [73] [74]. The former uses dynamic slicing for statistical bug localization in RTL. The latter proposes dynamic slicing and location-ranking-based method for accurately pinpointing the error locations combined with a dedicated set of mutation operators.

However, different from the works listed above, Chapter 5.3 proposes a dynamic slicing-based technique that implicitly covers the golden run fault collapsing, thereby significantly speeding up the fault injection process.

Moreover, relevant solutions [75] [76] are available in the context of the proposed solutions described in Chapter 5.4. However, these state-of-the-art approaches rely on the static cones pre-analysis only and do not consider if a SET fault actually propagates to the FF inputs. [75] proposes an RTL fault injection model, which is representative for laser fault attacks. To do that, the authors analyze the circuits structurally and find intersection cones that guide the fault injection in advance. On the other hand, they neither create FF sets that cover all SET faults nor optimize FF sets by considering true/false paths. Similarly, [76] models the locality of a laser attack in case of multiple-bit faults. The authors analyze the circuits structurally as well and, afterward, create FF sets. However, the authors consider only the supersets and reject all the subsets. In this way, each combination of SEUs in the superset is a trial to hit a fault in any smaller cone intersection. Nevertheless, the probability of hitting a SET in case of any superset by selected random multiple SEU is low.

Other studies investigate the impact of SET faults. [77] estimates the impact of SET faults without layout information by identifying a pair of gates in which SET can propagate to multiple outputs. [78] analyzes the impact of SETs through Algebraic Decision Diagrams and Binary Decision Diagrams (BDD) and [79] improves this method by considering multiple effects. Finally, [80] suggests performing a stochastic gate-level simulation for small circuits. Last but not least, some works investigate the combination of different fault analysis technologies such as [81], and [82]. These works combine the strength of formal methods and fault injection simulators; however, they analyze only permanent faults and do not analyze the representation of gate-level SET faults at RTL.

Different from the works listed above, Chapter 5.4 proposes a more efficient technique to prune the fault space by considering the propagation of SET faults. The significant speedup is achieved by running the RTL fault injection procedure on multiple flip-flop upset faults accurately selected.

5.3 Dynamic Slicing based Fault List Pruning

In this chapter, we explain the simulation-based fault injection methodology based on Dynamic Slicing to minimize the number of fault injections. This work is published in **IV** and **VI**. The proposed methodology identifies critical faults that cause the system to fail in the absence of a safety mechanism and injects only critical faults during the transient simulation-based fault injection campaigns. Using critical faults to estimate diagnostic coverage of safety mechanisms or fault coverage of workloads eliminates the possibility of simulation-based fault injection experiments to produce no error [6]. The main contribution of this work is three-fold as follows:

- Dynamic slicing on HDL to generate critical fault list
- Implicit fault collapsing within the slicing model, meaning that the fault list obtained by the proposed slicing method has an additional feature of avoiding injections at time-steps as data inside registers is not being consumed
- Language-agnostic RTL simulation-based fault injection supported by industrial-grade EDA tool flow

The proposed flow is shown in Fig. 17 and starts with the (1) extraction of static slices for the target observation point. In parallel, hardware code coverage data is generated by (2) simulation-based code coverage analysis for the design with pre-defined stimuli in the testbench. Next, (3) the dynamic slicing procedure identifies the intersection of the identified static slice and covered code items and results in a set of clock-cycle-long dynamic slices for the given observation point. Finally, (4) the simulation-based fault injection selects critical faults from the dynamic slices, injects them at the specified time,

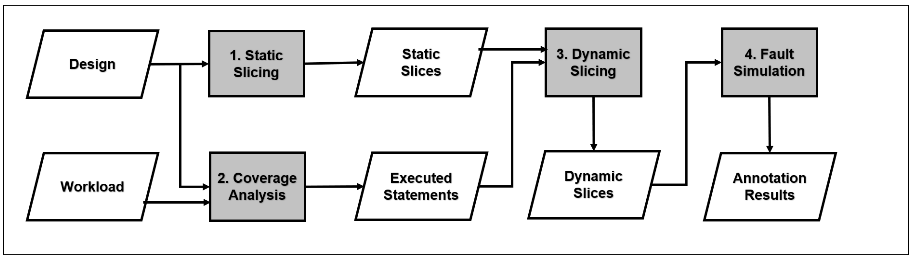


Figure 17: Dynamic Slicing based Fault List Pruning Flow [4] [6]

and evaluates the fault propagation. We explain the details of the proposed methodology in the following sub-chapters using a motivational example depicted in Fig. 18, i.e., a VHDL implementation of a signal *chopper* design [83].

5.3.1 Static Slicing

Static slice includes all statements that affect the value of a variable *v* for all possible inputs at the point of interest, e.g., at the statement *x*, in the program. In the RTL code, the static slice shows the dependency between HDL statements [84].

The simple design *chopper* in Fig. 18 has four outputs representing different chops for the input signal *SOURCE* based on the design configuration by inputs *INV* and *DUP*. This design makes it possible to perform a search backward to find dependencies in the HDL, which is a static slice of the selected output. Taking *TAR_F* as a considered output, the resulting static slice is computed as shown in Fig. 19 with the help of the formal analysis tool's structural analysis capability. Additionally, the column Static Slice in Fig. 18 marks HDL statements of the static slice of the *TAR_F* output. For instance, as the static slice of *TAR_F* does not include Line-40, *H0* is counted as outside of the static slice, and for the *TAR_F* output, there is no need to inject fault on *H0*. Hence, *H0* is not included in Fig. 19 as well.

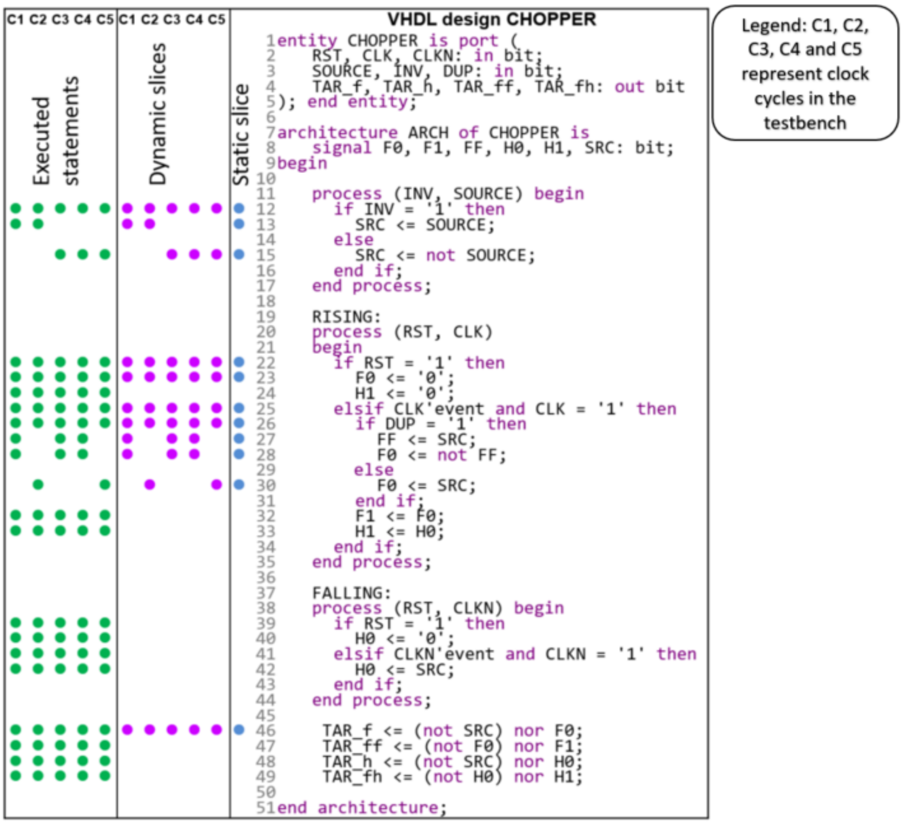
Furthermore, Fig. 18 implies that a static slice does not depend on clock cycles (represented as C1, C2, C3, C4, and C5 in Fig. 18) while executed statements and dynamic slice may change for each clock cycle (as explained in the following sub-chapters). In summary, a static slice of a considered output includes statically available information only as it does not make any assumptions on inputs.

After performing Static Slicing as the first step of fault list pruning, the next step is to run Coverage Analysis as explained below.

5.3.2 Coverage Analysis

In parallel to the Static Slicing step, the RTL design is simulated in the logic simulation tool to dump and analyze the hardware coverage data. In this step, we dump the coverage data for each clock cycle using a script set to find what statements in the RTL are executed for each clock cycle. In the proposed methodology, one clock cycle defines the size of the dynamic slice. We use a coverage tool and coverage metrics in order to find executed statements. After loading a simulation run into the coverage tool, coverage metrics data scored in that run can be analyzed.

In this chapter, we use hardware code coverage which measures how thoroughly a testbench exercised the lines of HDL code. Hardware code coverage includes block coverage, branch coverage, statement coverage, expression coverage, and toggle coverage. All these coverage types, except toggle coverage, can be used for the analysis presented in this chapter. Block coverage identifies the lines of the code that get executed during a simulation run. It helps to determine if the testbench executes the statements in a block. Additionally, branch coverage complements block coverage by providing more precise coverage results for reporting coverage numbers for various branches individually. Statement coverage is just a subset of block coverage, and it shows the execution of all the executable statements in the RTL. Finally, expression coverage provides information on why a conditional piece of code is executed. At the end of this step, we generate executed statements data to find dynamic slices in the next step. Fig. 18 shows executed statements for five clock cycles (C1, C2, C3, C4, C5).



```
entity CHOPPER is port (
    RST, CLK, CLKN: in bit;
    SOURCE, INV, DUP: in bit;
    TAR_f, TAR_h, TAR_ff, TAR_fh: out bit
); end entity;

architecture ARCH of CHOPPER is
    signal F0, F1, FF, H0, H1, SRC: bit;
begin

    process (INV, SOURCE) begin
      if INV = '1' then
        SRC <= SOURCE;
      else
        SRC <= not SOURCE;
      end if;
    end process;

    RISING:
    process (RST, CLK)
    begin
      if RST = '1' then
        F0 <= '0';
        H1 <= '0';
      elsif CLK'event and CLK = '1' then
        if DUP = '1' then
          FF <= SRC;
          F0 <= not FF;
        else
          F0 <= SRC;
        end if;
        F1 <= F0;
        H1 <= H0;
      end if;
    end process;

    FALLING:
    process (RST, CLKN) begin
      if RST = '1' then
        H0 <= '0';
      elsif CLKN'event and CLKN = '1' then
        H0 <= SRC;
      end if;
    end process;

     TAR_f <= (not SRC) nor F0;
     TAR_ff <= (not F0) nor F1;
     TAR_h <= (not SRC) nor H0;
     TAR_fh <= (not H0) nor H1;

end architecture;
```

Figure 18: Dynamic Slicing on a motivational example chopper [4] [6]

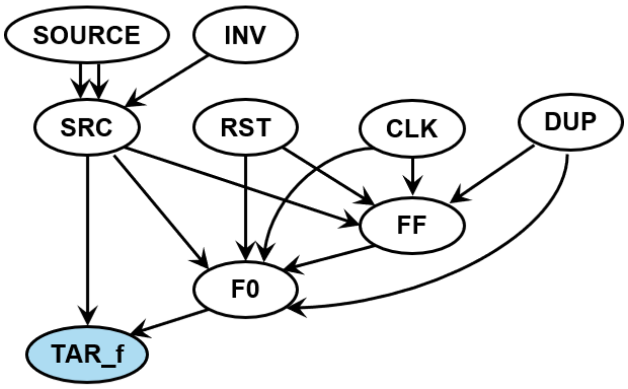


Figure 19: Backward static slice on the signal TAR_F in the chopper design [6]

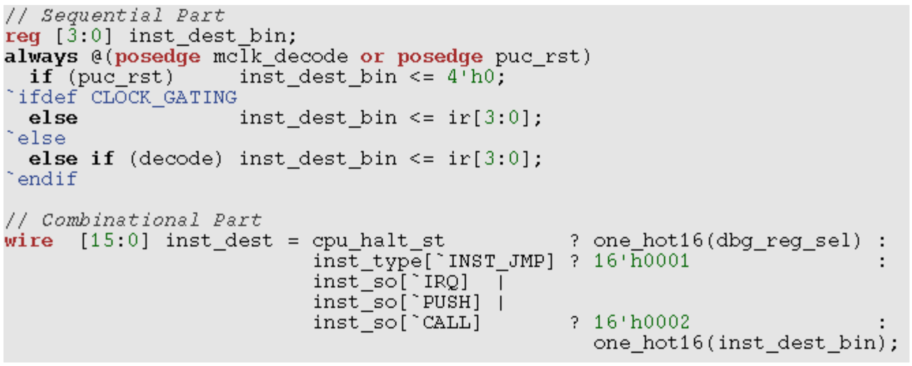
5.3.3 Dynamic Slicing

Dynamic slice includes those statements that actually affect the value of a variable *v* for a particular set of inputs of the RTL, so it is computed on a given input [85]. Thus, it provides more narrow slices than static slices and consists of only the statements that affect the value of a variable for a given input.

In a nutshell, a dynamic slice is the intersection of a static slice and executed statements. We illustrate the concept of the dynamic slice in Fig. 18. This figure also shows how dynamic slices narrow down the fault space compared to the state-of-the-art static slice approach. For example, during the time window C5, the register *FF* (Line-27) is not in the dynamic slice, meaning that there is no need to inject a fault in *FF* at the C5 time window. Therefore, the dynamic slice provides for critical faults and eliminates those faults that are not critical. In this way, it is possible to prune the fault list by injecting only critical faults. This provides a speed-up in the simulation-based fault injection time as each injected fault increased the total run time of a simulation-based fault injection campaign.

5.3.3.1 Implicit Fault Collapsing in Dynamic Slices

In this proposed methodology, dynamic slices cover both sequential and combinational parts. In this way, all faults outside of dynamic slices are 100% undetected and can be collapsed to exclude them from the fault list. When considering the average CPU time per simulation of one fault, an undetected fault spends more CPU time than a detected fault as the simulation-based fault injection for an undetected fault lasts until the end of the simulation. Hence, it is very effective to identify undetected faults without running simulation-based fault injection campaigns. In Fig. 20, the dynamic slice is built by considering the register *inst_dest_bin* and *inst_dest* (combinational) so that we can have 100% accurate results. This is called implicit fault collapsing since we avoid injections at time-steps when data inside registers are not being consumed.



```
// Sequential Part
reg [3:0] inst_dest_bin;
always @(posedge mclk_decode or posedge puc_rst)
  if (puc_rst)      inst_dest_bin <= 4'h0;
`ifdef CLOCK_GATING
  else              inst_dest_bin <= ir[3:0];
`else
  else if (decode) inst_dest_bin <= ir[3:0];
`endif

// Combinational Part
wire  [15:0] inst_dest = cpu_halt_st          ? one_hot16(dbg_reg_sel) :
                         inst_type[`INST_JMP] ? 16'h0001               :
                         inst_so[`IRQ]  |
                         inst_so[`PUSH] |
                         inst_so[`CALL]       ? 16'h0002               :
                                                one_hot16(inst_dest_bin);
```

Figure 20: Implicit fault collapsing [6]

5.3.4 Simulation-based Fault Injection

In this step of the proposed flow shown in Fig. 17, simulation-based fault injection is performed to verify the capability of a safety mechanism to recognize failures in a design's functionality by injecting faults into the design. To inject faults into a design, a fault simulator needs to know the fault target at which to inject a fault. In this chapter, we enable fault instrumentation on the dynamic slices, more specifically on registers that are in dynamic slices. In other words, the proposed methodology identifies critical faults from dynamic slices and injects them at specified times. As a fault model, a single-clock-cycle bit-flip fault within the RTL registers is used.

In brief, simulation-based fault injection is utilized to show the effectiveness of the proposed method. We inject one fault in one simulation run. Also, in the case of having more than one observation point in the campaign, the proposed method prevents multiple injections of faults within the overlap of static slices.

5.3.5 Experimental Setup

In order to verify the accuracy of the proposed fault list pruning technique, the experimental setup is developed as shown in Fig. 21. This setup works as an application aiming to automate the execution of simulation-based fault injection campaigns using the different tools. We create generic scripts to activate the tools and automate the flow. The proposed methodology is integrated into Cadence flow in this chapter, but it can be applied using tools by any major EDA vendor.

In the first step, a backward static slice is built for a selected observation point by using Cadence® JasperGold Formal Verification Platform. Then, we generate coverage results through Cadence® Xcelium™ for each clock cycle that defines the size of the dynamic slices. In the next step, static slice and executed statements data are sent to simulation-based fault injection to define the fault target for the campaign. Annotation results provide information regarding the number of injected faults, the number of detected and undetected faults. Moreover, we also use the profiling feature of the tool that measures where CPU time is spent during simulation. The profiler generates a run-time profile file that contains simulation run-time information that is useful for comparing the execution time of different campaigns. Cadence® Xcelium Fault Simulator is used for simulation-based fault injections.

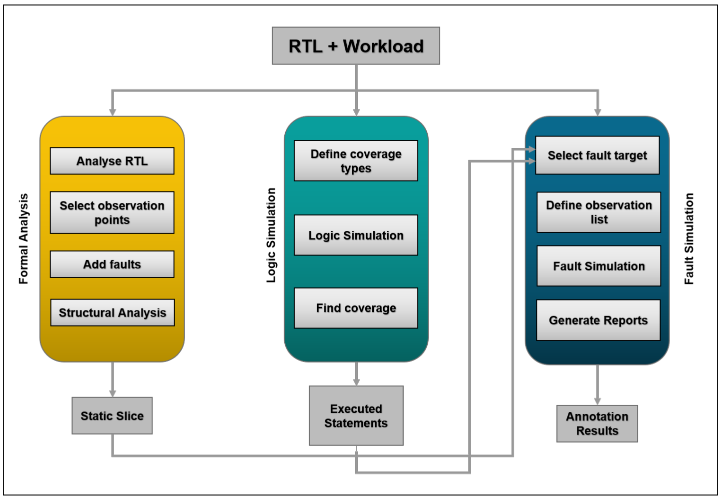


Figure 21: Overall flow of experimental setup [6]

5.3.6 Experimental Results

In this chapter, we evaluate the proposed methodology on a 16-bit microcontroller core *openMSP430* [86] with a single address space for instructions and data. To show the effectiveness of the proposed technique, Sandbox, Dhrystone, and Coremark workloads are run on the *openMSP430*. Additionally, we show the results in two categories as fault list reduction and time savings. We also evaluate the accuracy of this methodology by comparing the results to a state-of-the-art static slicing optimization method.

In the first step, the backward static slice is built from *dmem_din* observation point, which is the main output of the core, and then coverage data is calculated. Next, considering the registers in the static slice, the instruction source (*inst_src_bin*) and the destination register (*inst_dest_bin*) are selected as fault targets to apply the proposed method because these registers are widely used in simulation-based fault injection applications as they hold all instructions. Moreover, we perform a simulation-based fault injection campaign for each workload and a fault target (and) for each approach listed below:

- Static Slicing: This represents the state-of-the-art analysis.
- Dynamic Slicing: The proposed technique to prune the fault list.

Table 9 presents the results. Furthermore, Table 9 gives the number of Detected, Undetected, and Total faults as well as the CPU time of overall regression and fault coverage of given workload. For the execution of Dhrystone and Coremark workloads in Static Slicing, 100k faults are selected as fault samples after the warm-up phase of the CPU.

Table 9: Experimental Results in openMSP30 [6]

	Sandbox		**Dhrystone**		**Coremark**	
	Static Slicing	**Dynamic Slicing**	**Static Slicing**	**Dynamic Slicing**	**Static Slicing**	**Dynamic Slicing**
inst_dest_bin						
Detected	8036	8036	56236	56236	48891	48891
Undetected	3996	2852	43764	42404	51109	47809
Total	12032	10888	100000	98640	100000	96700
Total CPU time	1197.1s	994.7s	658919.7s	622459.0s	3437663.0s	3323109.9s
Fault Coverage	66.788%	73.806%	56.236%	62.735%	48.891%	50.559%
inst_src_bin						
Detected	2423	2423	34766	34766	45161	45161
Undetected	9609	8413	65234	63498	54839	48051
Total	12032	10836	100000	98264	100000	93212
Total CPU time	1488.2s	1284.2s	803009.1s	790300.0s	3575198.1s	3178378.2s
Fault Coverage	20.137%	22.361%	34.766%	35.380%	45.161%	48.450%

5.3.6.1 Fault List Reduction

Fig. 22 highlights reduction in the Total number of faults given in Table 9. All detected faults seen in Fig. 22 are critical faults. As seen in these charts, dynamic slicing is more effective in pruning the fault list than static slicing. Furthermore, Table 10 shows the percent reduction in the number of fault injections. The best reduction is achieved in Sandbox workload as 9.94%. The magnitude of the fault list reduction depends on the workload characteristics. In these experimental results, the fault list reduction varies between 1.36% and 9.94%. This analysis reveals that dynamic slicing prunes the fault list and successfully identifies the critical faults while analysis and optimization effort costs are minor. Additionally, identifying undetected faults and excluding them from the fault list provides increased fault coverage, as can easily be seen in Table 9.

Table 10: Percentage of Reduction of the Total Number of Injections with Dynamic Slicing [6]

	Sandbox	Dhrystone	Coremark
inst_dest_bin	9.51%	1.36%	3.3%
inst_src_bin	9.94%	1.74%	6.79%

5.3.6.2 Time Savings

Table 9 shows the total CPU time of overall regression for each simulation-based fault injection campaign. Time savings are highlighted in Table 11. As it can be seen, dynamic slicing provides various time savings from 1.58% to 16.91%. As in fault list reduction, time

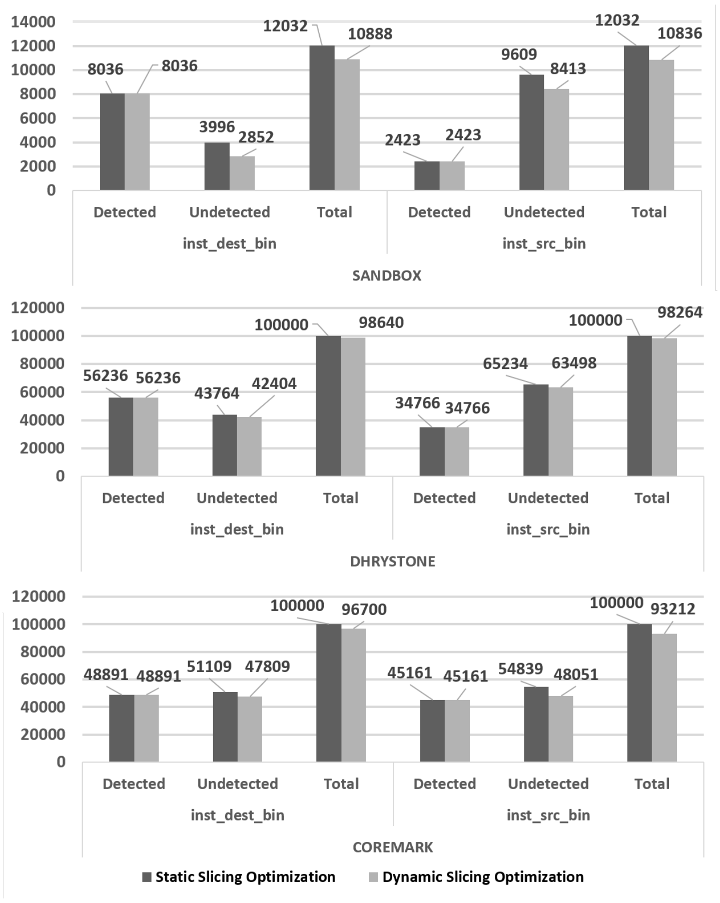


Figure 22: Fault list reduction based on three workloads [6]

savings depend on the workload characteristic. When considering the need for multiple simulation-based fault injection campaigns in real-life applications with thousands of fault targets and clock cycles, this time savings can expeditiously increase.

Table 11: Time Savings Using Dynamic Slicing [6]

	Sandbox	Dhrystone	Coremark
inst_dest_bin	16.91%	5.53%	3.33%
inst_src_bin	13.71%	1.58%	11.10%

5.3.6.3 Accuracy

This chapter shows the results of a fault injection campaign performed using dynamic slicing, along with a state-of-the-art static slicing approach. These results reveal that dynamic slicing achieves the same number of detected faults as static slicing campaigns. This means that dynamic slicing can be used for different purposes as it is an accurate fault injection methodology. For instance, SFI [87] prunes the fault list in terms of margin of error with a given confidence level. However, dynamic slicing excludes only non-critical faults and finds all critical faults with a 100% accuracy.

5.3.7 Conclusions

Simulation-based fault injection on RTL requires an excessively long simulation time, which prevents detailed reliability evaluation of hardware components with significant injections. This chapter presents a methodology to speed-up simulation-based fault injection campaigns by minimizing fault injection locations. The methodology applies dynamic slicing on HDL to accurately pinpoint fault injection locations and allows injection of critical faults in these time windows. In this way, this chapter narrows down the fault space and provides reduced simulation time. Moreover, an average 5-10% extra gain in simulation time for simulation-based fault injection significantly improves the total chip validation costs, as this phase is the most time-consuming. The proposed method is language-agnostic and suitable for industrial grand EDA tool flows. Experimental results on industrial-size CPU show that we achieve significant speed-up of the simulation-based fault injection compared to the state-of-the-art flows.

5.4 Mapping based Fault List Pruning

One of the challenges of simulation-based fault injection campaigns is the vast number of possible fault locations because engineers simulate a fault-free design and its copies with faults injected one at a time. This may imply enormous execution times, especially for the simulation-based fault injection at the gate-level. Hence, there is a high demand for methodologies that can support designers in the early-stage design exploration of reliability factors. Moreover, simulation-based fault injection into gate-level models is quite late in the IC development cycle, and design modifications become more expensive in terms of the required engineering effort. Therefore, early design evaluation is necessary in both safety and security-related applications to minimize design iterations and resources, thus enabling faster design closure times.

In this chapter, we focus on SET faults at the gate-level and propose an efficient solution to represent them by multiple SEU faults at the RTL [9]. The relevance of this problem for safety-critical applications grows with the downscaling of the technology nodes, forcing designers to evaluate the system's safety against SET faults, which affect combinational elements of the circuit. However, this comprehensive evaluation at the gate-level is not affordable regarding the execution time of fault injection campaigns for industrial-sized designs. Furthermore, from the security point of view, SET faults at the gate-level represent laser fault attacks, which can be observed in flip-flops (FFs) as single or multiple errors [88]. Here, it is crucial to evaluate laser attacks to determine which vulnerable SET faults create single or multiple errors in the sequential elements of the design.

We propose a methodology for representing gate-level transient faults, such as SETs, by Multiple Flip-Flop Upset (MFFU) at RTL to tackle the listed problems. In the case of Soft Error Reliability (SER) assessment for safety applications such as automotive, MFFU becomes functionally equivalent for EDA tools to multiple simultaneous SEUs. For vulnerability analysis against fault-injection attacks on security-critical designs, MFFU refers to single and multi-bit fault injections. In this chapter, first, we identify the static fan-in cones of each flip-flop at the gate-level. Second, we perform propagation analysis to identify SET faults with true (sensitizable) paths to FF inputs. In this way, we obtain optimized FF sets as representatives of all SET faults to guide RTL multiple SEU fault injection campaigns. As a result, this method can successfully reduce the fault space and enhance the high complexity of simulation-based fault injection campaigns. Without loss of generality, the proposed methodology is demonstrated on a Cadence EDA (Electronic Design

Automation) tool flow, but it remains applicable to other tool flows as well. The main contribution of this work is as follows:

- An approach to move the gate-level SET vulnerability analysis to RTL
- A technique to reduce the fault space at RTL by applying gate-level propagation analysis
- A systematic and workload-independent methodology for representing the gate-level SETs by multiple SEUs at RTL supported by industrial-grade EDA tool flow

5.4.1 The Proposed Methodology

This chapter aims to identify MFFU sets for RTL simulation-based fault injection, which represent all gate-level SET faults. By doing so, we reduce the number of injections required to evaluate the effect of SET faults.

The SET fault model implies flipping the value of a signal in the combinational cloud and holding the value for a specified period of time. SEU fault model implies flipping the value of the output of a sequential element and holding it until it is overwritten with new data. Therefore, SEUs can be applied to the outputs of sequential elements, such as memories, flip-flops, and latches. We apply SET faults for one clock cycle length. The proposed flow is shown in Fig. 23 and starts with the (1) extraction of static fan-in cones of each flip-flop in the gate-level netlist. In the next step (2), flip-flop sets are created to represent each SET fault on flip-flops' fan-in cones. Then, we perform propagation analysis (3) to check if SET faults propagate to the flip-flop inputs. If a SET fault does not propagate, then we check if these changes created flip-flop sets. In this way, we obtain optimized flip-flop sets, which are representative of all SET faults, which propagate to the flip-flop inputs. Finally (4), we calculate the fault space to see the reduction compared to state-of-the-art and random multi-bit injection approaches. The following sub-chapters explain each step of the proposed method in detail.

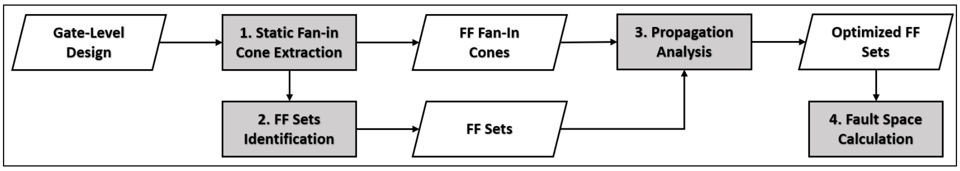

Gate-Level Design
1. Static Fan-in Cone Extraction
FF Fan-In Cones
3. Propagation Analysis
Optimized FF Sets
2. FF Sets Identification
FF Sets
4. Fault Space Calculation

Figure 23: Steps of Mapping based Fault List Pruning Methodology [9]

5.4.2 Static Fan-in Cone Extraction of Flip-Flops at gate-level

As a first step, we extract fan-in cones of each flip-flop at the gate-level, as it is illustrated in Fig. 24. In the beginning, we generate a list of all faults in the design. Then, we extract fan-in information from all flip-flops in the ingress combinational part of the design. Each fan-in cone search starts from a flip-flop and expands backward, i.e., in the direction of inputs of the combinational cloud, until it encounters a FF output or a primary input (PI). Finally, all SETs in each cone are enumerated to map each SET to a flip-flop set. This step is performed by using Cadence® JasperGold Functional Safety Verification App.

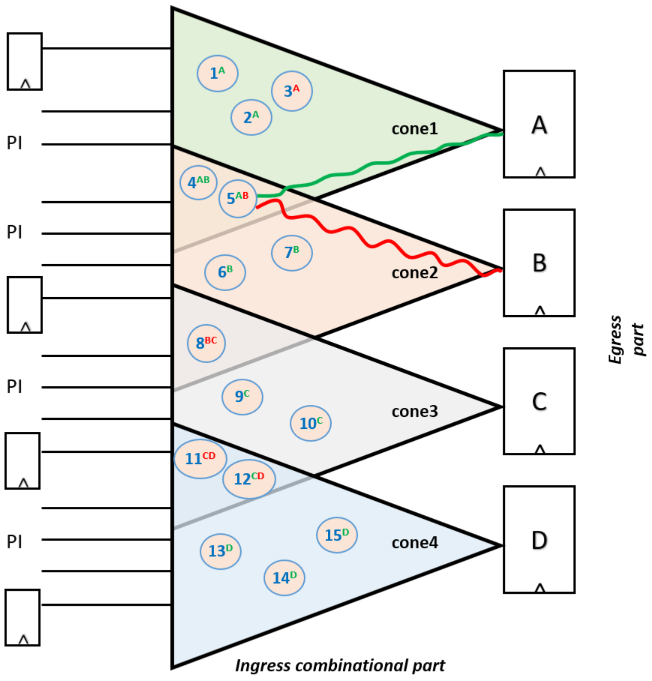


Figure 24: Extracting fan-in cones of each FF and finding propagation paths [9]

5.4.3 Flip-Flop Sets Identification

The second step of the proposed methodology is identifying flip-flop sets, which will be used as an MFFU injection target in the following steps. To do that, we consider each fan-in cone independently and determine flip-flop sets, which cover all possible scenarios, as shown in the second column of Table 12. For instance, if cone-1 is affected by a SET fault, we can cover this SET fault by injecting multiple MFFUs on A and B because cone-1 has an intersection with cone-2, which is the fan-in cone of B. This process is repeated for each cone, and FF sets are obtained with a size between 1 (in case the cone does not intersect with any other cones) and N FFs (in case all cones have an intersection).

Extracted FF sets are flip-flops of the circuit potentially affected by a SET. Therefore, MFFU injection can be limited to this set of FF. Table 12 also shows the multiplicity information of each FF set. The multiplicity of a FF set is the number of FF in a set. For instance, if a SET fault occurs in cone-2, it can propagate to the A, B, C FFs, causing different combinations of upsets on this set. This means that the less is the number of FF in a set (less multiplicity), the higher is the probability of hitting a real MFFU. We will use this information in the following steps. Moreover, multiplicity is important for calculating fault space, which will be given in the next sections. It is obvious that there are 8 combinations in one FF set with a multiplicity 3.

5.4.4 Propagation Analysis

In this chapter, unlike state-of-the-art research, we also take propagation of faults into consideration to reduce fault space more. For this step, we deploy the formal techniques to investigate the behavior of a design under fault. The theory behind formal techniques is creating a Boolean function representation of a design under test so that formal proof can

Table 12: Results of Example Design Given in Fig. 24 [9]

Affected Cone	FF Sets	Multiplicity	Optimized FF Sets	Optimized Multiplicity
Cone 1	A, B	2	A, B	2
Cone 2	A, B, C	3	A, B	2
Cone 3	B, C, D	3	C	1
Cone 4	C, D	2	D	1

be used. In order to achieve better performance in the modern formal tools, BDDs [89] and Multiway Decision Graphs (MDGs) [90] are widely used.

The formal analysis deploys formal methods to determine the propagation of faults. Propagation analysis verifies if there is a combination of inputs that provoke fault propagation. For example, if a fault propagates to flip-flop inputs, we accept that the fault has a true path to flip-flop inputs. Otherwise, it has a false path and should be excluded from the analysis. In this step, formal properties to perform the analysis are automatically generated and verified with respect to all possible input stimuli.

The simple and high-level example in Fig. 24 illustrates that there are some SET faults in the intersection cones with a false path to the FF inputs. In this figure, green paths and superscripts point to the true paths (fault propagates), while red ones show that the related fault has a false path (fault does not propagate). As a result of this step, we obtain optimized FF sets, as shown in the fourth column of Table 12. It is obvious that some larger FF sets are disappeared due to non-observable faults that cannot be propagated. In this way, optimized multiplicities are obtained along with the reduced number of flip-flop sets in some circuits. This step is performed by using Cadence® JasperGold Functional Safety Verification App. In the following sub-chapter, we show a more detailed motivational example for the propagation analysis.

5.4.4.1 Motivational Example: Removing the paths which cannot be propagated

To explain the propagation analysis in detail, we use a motivational example given in Fig. 25 which has fan-out nodes. The circuit includes an input **x**, and outputs of the gates **AND1**, **OR1** and **OR2**. The SETs may be simulated only for these fan-outs. The steps of the approach can be listed as follows:

- Static fan-in cone analysis gives us the following flip-flop sets of MFFU faults: (1, 2, 3, 4) for **x**, (1, 2, 3, 4) for **AND1**, (1, 2) for **OR1**, (2, 3) for **OR2**.

- After removing duplicated sets, we get the initial sets of MFFU faults: (1, 2, 3, 4), (1, 2), (2, 3).

- By propagation analysis, we see that for SET on AND1 we never reach all flip-flops, rather only either (1, 4) or (2, 3) due to the fact that the propagation of a SET at **AND1** is controlled by signal **x**=0 (by blocking two of four AND gates). Therefore, the superset (1, 2, 3, 4) for **AND1** should be replaced by subsets (1, 4) and (2, 3). In other saying, SET(**AND1**) is mapped to (1, 4) and (2, 3) flip-flop sets.

- Moreover, the SET on the input **x** is always blocked either on **AND5** (if output of **AND1**=1), or on **AND2**, **AND3**, **AND4** (if output of **AND1**=0). Hence, the superset (1, 2, 3, 4) for SET(x) should be replaced by (1, 2, 3).

- As a result, we get instead of initial (1, 2, 3, 4), (1, 2), (2, 3), optimized FF sets (1, 2), (2, 3), (1, 4) (2, 3), (1, 2, 3), where (2, 3) can be removed as it is duplicated.
- Thus, the final optimized FF sets: (1, 2), (2, 3), (1, 4), (1, 2, 3).

In this motivational example, we analyze the propagation of SETs only on **x** and the outputs of **AND1**, **OR1** and **OR2**. The propagation analysis is sufficient for the SETs at these four locations, representing the remaining SET faults in the fan-out free regions.

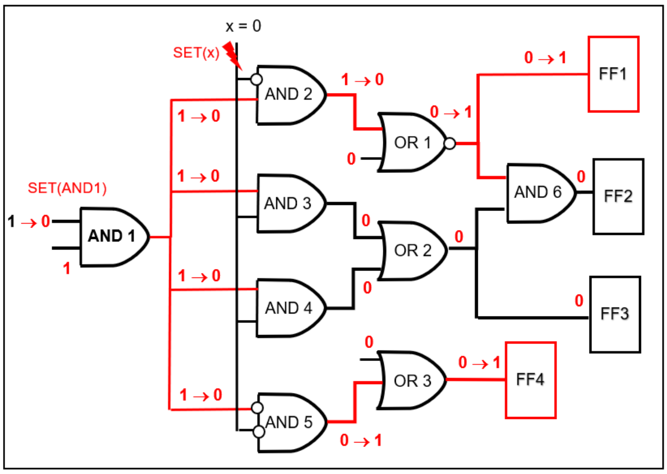


Figure 25: Motivational example to find propagated and not-propagated faults [9]

5.4.5 Fault Space Calculation

In the fault injection procedure, SEUs are injected in all possible locations and at each clock cycle [6]. Therefore, the number of injections required for a single transient fault is enormous, especially for industrial-sized designs. Therefore, optimization methods should be applied when considering the gate-level's size and the low speed of fault injection simulations. Hence, considering the vast number of SET injections at the gate-level, the proposed method in this chapter significantly reduces the number of injections by identifying optimized flip-flop sets compared to state-of-the-art, and random multi-bit injection approaches applied in safety and security applications.

The proposed methodology in this chapter can significantly reduce the fault space by leveraging the flip-flop sets with propagation analysis. In this chapter, we compare the results with the state-of-the-art and random multi-bit injection. State-of-the-art researches such as [75] and [76] rely on only a static approach and do not consider the propagation analysis. Similarly, the random multi-bit injection method considers all possible flip-flop combinations. In order to calculate fault space or the number of injections, we use the following equation where N is the number of flip-flops, k_1, k_2, ..., k_N are the numbers of FF in each set and $1 \leq k_i \leq N$. given in [76].

$$FaultSpace_{Total} = \sum_{i=1}^{N} (2^{k_i} - 1) \tag{6}$$

By using the above equation, the total fault space for the example given in Fig. 25 can be calculated effortlessly. As it is explained in Chapter 5.4.4.1, we have the initial and not-optimized sets which represent the state-of-the-art approach as (1, 2, 3, 4), (1, 2), and (2, 3). Therefore, by using the given formula, the total number of faults is 21. On the other hand, we have optimized flip-flop sets as (1, 2), (2, 3), (1, 4), (1, 2, 3), which require 16 injections. Therefore, the proposed method can reduce the total fault space from 21 to 16 for the motivational example given in Fig. 25.

5.4.6 Experimental Results

In order to verify the effectiveness of the proposed methodology, we evaluate the methodology on the ITC'99 [91] benchmark circuits.

In order to perform fan-in cone analysis and propagation analysis, we deploy Cadence tools along with the developed script sets, which execute on gate-level design. Meanwhile, all applied methods remain applicable to other tool flows. In the beginning, we synthesize Verilog or VHDL design through Cadence® Genus™ Synthesis Solution to obtain the gate-level representation of the design. Then, steps 1, 2, and 3 shown in Fig. 23 are performed on my application which deploys Cadence® JasperGold Functional Safety Verification App.

We use three methods to show the fault space reduction and compare the results. The first method is "without propagation analysis" which represents the state-of-the-art as in [76]. The main difference between our proposed methodology "with propagation analysis" and the state-of-the-art is the identification of true (sensitizable) paths. We leverage the analysis by identifying SET faults which do not propagate to flip-flop inputs so that fault space is reduced more. In other words, we cut down the pessimism in the results. The third approach used for comparison is "Random Multi-Bit injection". This is basically injecting faults on all possible combinations of flip-flops randomly that naturally cause huge fault space. The proposed application is capable of building the fault space for each method and given design without any significant effort.

All experimental results are presented in Table 13. The selected designs include various designs from the ITC'99 benchmark. During the creation of FF sets, we remove faults on the clock and reset signals from the analysis due to the fact that the clock tree is not known in this stage of the design. Other faults except clock and reset are kept as they are. This step is done in our application automatically. We show the number of sets, supersets, maximum multiplicity, and calculated fault spaces for each analysis and design. The number of sets shows the number of all identified FF sets before duplicated ones are removed. In contrast, the number of supersets points the same after duplicated ones are removed. Total Faults are calculated by using Equation 6.

In Table 13, it can be seen that the proposed methodology reduces the Total Faults significantly when compared to both state-of-the-art and the random multi-bit injection approaches. For some circuits such as **b01** and **b08**, we am able to reduce only the number of supersets while the maximum multiplicity is still the same in both cases. Moreover, there is no optimization achieved in **b06** due to the structure of the design. For the rest of the circuits given in Table 13, we both optimize the number of supersets and maximum multiplicity. Thereby, the total set of faults are optimized significantly, as shown in Fig. 26 (values are normalized). It is observable that total faults in the proposed methodology (light gray bars) are less than the other two methods. We also add that we reduce the fault space from 1.20 times to a few hundred times when compared without propagation analysis, depending on the circuit.

Moreover, we also compare our results with the well-known Statistical Fault Injection (SFI) approach [87] in case initial population sizes calculated before are used. SFI can be used for transient fault injection campaigns to reduce the execution times while keeping a meaningful number of injections with an error margin. This is one of the possible

Table 13: Experimental Results: Fault Space Achieved by Three Methods [9]

		without propagation analysis				with propagation analysis				Random Multi-Bit Injection
Circuit	**# FF**	**# sets**	**# superset**	**max multiplicity**	**# Total Faults**	**# sets**	**# superset**	**max multiplicity**	**# Total Faults**	**# Total Faults**
b01	5	5	2	4	1.80E+01	3	1	4	1.50E+01	3.20E+01 - 1
b02	4	4	1	3	7.00E+00	1	1	2	3.00E+00	1.60E+01 - 1
b03	30	8	3	12	4.14E+03	3	1	9	5.11E+02	1.07E+09 - 1
b04	66	27	10	19	4.00E+06	5	4	8	1.02E+03	7.38E+19 - 1
b05	34	62	2	33	9.00E+09	61	5	31	2.00E+09	1.72E+10 - 1
b06	8	7	5	4	4.30E+01	7	5	4	4.30E+01	2.56E+02 - 1
b07	46	51	2	35	4.00E+10	43	3	26	8.00E+07	7.04E+13 - 1
b08	21	19	2	18	2.70E+05	11	2	18	2.62E+05	2.10E+06 - 1
b09	28	14	1	28	3.00E+08	7	1	27	1.00E+08	2.68E+08 - 1
b10	17	45	9	11	5.91E+03	13	4	11	2.62E+03	1.31E+05 - 1
b11	31	43	9	18	4.65E+05	9	2	16	6.60E+04	2.15E+09 - 1
b13	50	40	13	13	9.15E+03	20	9	9	9.47E+02	1.13E+15 - 1

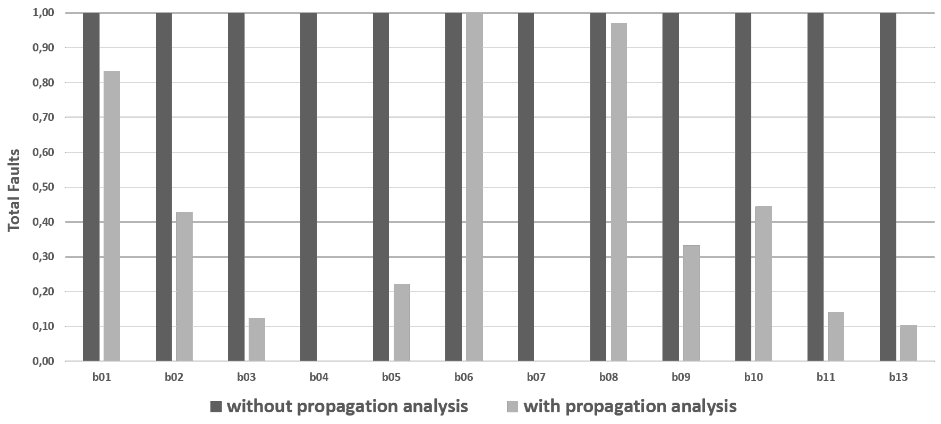


Figure 26: Fault space comparison [9]

ways to perform RTL fault injection campaigns after flip-flop sets are defined by using the methodology presented in this chapter. In an SFI campaign, the sample size or the margin of the error with a certain confidence level is determined by using the Equation 7 defined in [87]. In this way, it is possible to obtain precise results while injecting a small number of faults [87]. Moreover, the technique allows knowing the margin of error while restricting the campaign time to the minimum. To sum up, there are three confidence levels in SFI as 90%, 95%, and 99.8%. In this chapter, we only use the 95% confidence level as it is the one that is practically used in the industry. Also, three error margins are defined as 5%, 1%, and 0.1%.

$$n = \frac{N}{1 + e^2 \times \left(\frac{N-1}{t^2 \times p \times (1-p)}\right)} \tag{7}$$

In Table 14 and Table 15, we show the SFI results. The former details the comparison of "with propagation" and "without propagation" analyses, and the latter table gives the fault space when Random Fault Injection is performed. In this tables, N shows the initial population. In the presented case, N is equal to the total faults shown in Table 13. Moreover, n(5%), n(1%) and n(0.1%) show the required sample size with the error margins 5%, 1% and 0.1% respectively. This shows that the proposed methodology can prune the fault space from 1.12 times to a few hundred times if faults are injected using SFI. Note, the results for some sample sizes remain similar due to the fact that the initial population is always finite. Even so, we show that a significant reduction is achieved by using the proposed methodology, especially when we reduce the error margins. Therefore, it is efficient to use the proposed methodology and select a sample for fault injection among the pre-defined initial populations in the MFFU space identified using the method "with propagation analysis".

Table 14: Fault Space Comparison of "with propagation" and "without propagation" analyses with 95% Confidence Level [9]

	without propagation analysis				**with propagation analysis**			
Circuit	**N**	**n (5%)**	**n (1%)**	**n (0.1%)**	**N**	**n (5%)**	**n (1%)**	**n (0.1%)**
b01	1.80E+01	1.70E+01	1.80E+01	1.80E+01	1.50E+01	1.40E+01	1.50E+01	1.50E+01
b02	7.00E+00	7.00E+00	7.00E+00	7.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00
b03	4.14E+03	3.52E+02	2.89E+03	4.12E+03	5.11E+02	2.20E+02	4.85E+02	5.11E+02
b04	4.00E+06	3.84E+02	9.58E+03	7.74E+05	1.02E+03	2.79E+02	9.22E+02	1.02E+03
b05	9.00E+09	3.84E+02	9.60E+03	9.60E+05	2.00E+09	3.84E+02	9.60E+03	9.60E+05
b06	4.30E+01	3.90E+01	4.30E+01	4.30E+01	4.30E+01	3.90E+01	4.30E+01	4.30E+01
b07	4.00E+10	3.84E+02	9.60E+03	9.60E+05	8.00E+07	3.84E+02	9.60E+03	9.49E+05
b08	2.70E+05	3.84E+02	9.28E+03	2.11E+05	2.62E+05	3.84E+02	9.27E+03	2.06E+05
b09	3.00E+08	3.84E+02	9.60E+03	9.57E+05	1.00E+08	3.84E+02	9.60E+03	9.51E+05
b10	5.91E+03	3.61E+02	3.66E+03	5.88E+03	2.62E+03	3.35E+02	2.06E+03	2.62E+03
b11	4.65E+05	3.84E+02	9.41E+03	3.14E+05	6.60E+04	3.82E+02	8.39E+03	6.18E+04
b13	9.15E+03	3.69E+02	4.69E+03	9.06E+03	9.47E+02	2.74E+02	8.62E+02	9.46E+02

Table 15: Fault Space in the case of Random Fault Injection with 95% Confidence Level [9]

	Random Multi-Bit Injection			
Circuit	**N**	**n (5%)**	**n (1%)**	**n (0.1%)**
b01	3.20E+01 - 1	3.00E+01	3.20E+01	3.20E+01
b02	1.60E+01 - 1	1.50E+01	1.60E+01	1.60E+01
b03	1.07E+09 - 1	3.84E+02	9.60E+03	9.60E+05
b04	7.38E+19 - 1	3.84E+02	9.60E+03	9.60E+05
b05	3.44E+10 - 1	3.84E+02	9.60E+03	9.60E+05
b06	2.56E+02 - 1	1.54E+02	2.49E+02	2.56E+02
b07	7.04E+13 - 1	3.84E+02	9.60E+03	9.60E+05
b08	2.10E+06 - 1	3.84E+02	9.56E+03	6.59E+05
b09	2.68E+08 - 1	3.84E+02	9.60E+03	9.57E+05
b10	1.31E+05 - 1	3.83E+02	8.95E+03	1.15E+05
b11	2.15E+09 - 1	3.84E+02	9.60E+03	9.60E+05
b13	1.13E+15 - 1	3.84E+02	9.60E+03	9.60E+05

5.4.7 Conclusions

This chapter proposes a methodology to represent gate-level SET faults by multiple SEU faults at RTL. It enables a solution for the high complexity problem of expensive gate-level fault injection campaigns by changing the abstraction level. We improve the state-of-the-art by considering the propagation analysis of each SET fault. First, we find static fan-in cones of each flip-flop at the gate-level. Second, flip-flop sets are created pessimistically, meaning that propagation analysis is not considered. Third, we execute propagation analysis using a formal approach to find SET faults that propagate to flip-flop inputs. Then, optimized flip-flop sets are created again with less pessimism. Finally, we calculate the fault space to show the effectiveness of the proposed methodology. In this way, we significantly reduce the number of fault injections and obtain a higher probability of hitting a true multiple SEU fault. Experimental results show that we make the fault space smaller by up to tens to hundreds of times.

5.5 Chapter Conclusions

Simulation-based fault injection is an adopted technique for analyzing complex designs used in safety or security critical applications. However, due to the complexity of hardware and software structure of these applications, simulation-based fault injection campaigns require huge execution times. In addition, a transient fault injection campaign is much more challenging as engineers need to inject a fault at each fault location and clock cycle. Therefore, there is a high need to accelerate and develop techniques to optimize these campaigns. One way of doing this is to prune the fault list so that engineers can avoid unnecessary injections.

In this chapter, we present two techniques that prune the transient fault list. First is Dynamic Slicing-based Fault List Pruning. This technique generates a critical fault list by selecting critical time-steps at which transient faults are injected. It deploys static slicing, coverage analysis, dynamic slicing to prune the fault list. Experimental results on an industrial CPU demonstrate that the proposed technique can provide 5-10% extra gain in execution time for simulation-based fault injection by keeping the accuracy. The second is Mapping based Fault List Pruning. This technique changes the abstraction level from gate-level to RTL by mapping SET faults to multiple SEU faults. Hence, only an affordable number of multiple SEU injections are performed instead of injecting millions of SET faults. Experimental results demonstrate the feasibility of the proposed technique to successfully reduce the fault space and also its advantage with respect to the state-of-the-art. Furthermore, it is shown that the approach is able to reduce the fault space, and therefore fault injection effort, by up to tens to hundreds of times.

In conclusion, the presented techniques improve state-of-the-art by reducing fault space of transient fault injection. This is necessary to avoid an enormous number of injections in modern complex designs used in critical safety and security areas.

6 Enhancing Hardware Fault Classification using Formal Analysis

In this chapter, we explain a technique to enhance hardware fault classification for automotive ICs using formal analysis. The results of this chapter were published in **VIII** and **X**.

6.1 Introduction

Concerning automotive applications, each electronic system must detect and correctly manage a high percentage of potential faults during the operation in the field to avoid life-critical situations. In order to decide which faults could disturb the safety-critical functionality of an IC, faults must be classified based on their effects in the operation mode using expert judgment and a combination of tools. Two fault sub-classes are identified, named *safe* and *dangerous* faults. A *safe* fault does not contribute to the violation of the safety goal, whereas a *dangerous* fault may lead to a failure relevant for the overall system, that is, create a hazard. Examples of safe faults include faults located in parts of an IC that are not used by the application and faults masked by some mechanism. Fault classification is of prime importance for the test of ICs in the operational mode. This test can be performed resorting to different solutions, including Design for Testability (e.g., BIST) and STLs based on the SBST paradigm [92]. In both cases, identification of safe faults is vital since it enables to remove safe faults from the initial (normally huge) fault list and to focus the test efforts towards the remaining faults, i.e., the testable ones [93]. Identifying safe faults thus makes it easier to reach the target diagnostic coverage (DC), helping to achieve safety requirements, such as a higher automotive safety integrity level (ASIL) [12]. For these reasons, there is a high demand for an automated, systematic, and comprehensive safe fault identification technique.

The effects of a fault classification flow are summarized in Fig. 27 referring to a generic case study. We assume that an SoC runs a single software (SW) application during its operational life and uses an STL as a safety mechanism. Therefore, the DC of this STL must be calculated to prove that it detects dangerous faults up to a certain extent in the target design. In the first step of the flow, without any classification, all the faults are unknown, as shown in Fig. 27. Then, an *Initial Classification* is performed to identify the first group of structurally-safe faults, i.e., those which are safe due to the IC structure (e.g., faults located on lines which are not connected to the IC Primary Inputs and/or Outputs). These kinds of safe faults can be identified using any Automatic-Test-Pattern-Generation (ATPG)

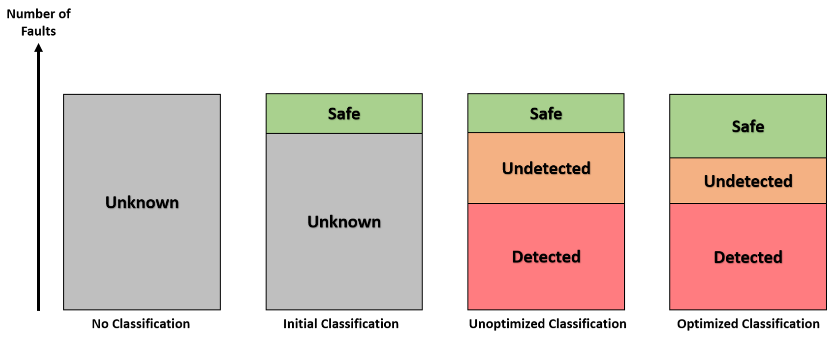


Figure 27: Hardware Fault Classification Flow [10]

or formal analysis tool. However, other safe faults may exist, which cannot be identified by these tools; therefore, a considerable amount of faults are still unknown after the first step. The unknown faults need to be further analyzed to check whether their effects may impact the safety-critical functionalities or not. Thus, simulation-based fault injection with an STL is deployed to classify faults better. In practice, this step (named *Unoptimized Classification* in Fig. 27) produces inaccurate results since it is often impossible to exhaustively evaluate all possible input stimuli or activate all possible operating modes in an application or system [41]. Undetected faults may correspond either to *safe* or *dangerous* faults. As in Fig. 27, simulation-based fault injection targets unknown faults and classifies them as either detected or undetected based on the propagation of faults. A non-negligible amount of undetected faults may be observed depending on the workload that runs on the target design. Usually, all the undetected faults are pessimistically classified as dangerous. For this reason, the gathered figures from simulation-based fault injection may not be representative of the design operational behavior as not all faults can be accurately classified. DC is calculated in this step using (8) where Detected is the number of faults classified as detected and dangerous by simulation-based fault injection, Total is the size of the target system's fault list, and Safe is the number of safe faults. The purpose is to check if the collected results from simulation-based fault injection satisfy the desired safety metrics. If DC is not enough, the test must be improved, or an additional classification effort targeting undetected faults, i.e., a subset of undetected faults, are required to classify their effects. Experts usually perform this step based on their design knowledge; however, this is error-prone and time-consuming. Consequently, the *Unoptimized Classification* implies that there is still room for improvement in the fault classification pessimism. Finally, using the technique presented in this work, a formal analysis approach optimizes the fault classification (named *Optimized Classification*) as shown in the fourth bar of Fig. 27, which targets the identification of more safe faults reducing the number of undetected faults and, therefore, the overall pessimism of the classification. The *Optimized Classification* decreases the denominator of (8) by classifying more safe faults than in the *Unoptimized Classification*, and the DC is increased.

$$DC = Detected/(Total - Safe). \tag{8}$$

This chapter advances hardware fault classification with an automated workflow, which assists safety experts in addressing fault classification reducing human error and the time to signoff. The present work focuses on the automated analysis of undetected faults to check whether they affect the safety critical functionalities of ICs. In the case that a fault cannot violate a safety goal or disturb safety critical outputs, it is defined as a safe fault. We consider a realistic scenario corresponding to a special-purpose system, i.e., an SoC which performs a single SW application, which remains the same during the whole operational life. Using the proposed technique, we can identify application-dependent safe (App-Safe) faults. One example of App-Safe faults is associated with the faults in the CPU debug unit, which is not used by the SW application during the operation life of the SoC. For this purpose, first, we perform several logic simulations to extract a target system's operational behavior by investigating code coverage results. Then, the candidates for being labeled safe faults which are not safety related are automatically translated into formal properties, which then configure the formal environment to identify App-Safe faults.

As a case study, the AutoSoC benchmark [7], an automotive representative SoC, and the cruise-control-application (CCA) as a target SW application are used. we focused on the CPU core and several peripherals, i.e., the universal asynchronous receiver–transmitter IP (UART) and the controller area network controller IP [53].

This chapter addresses the problem of what is new in ISO 26262 functional safety verification that differs from general reliability in terms of safe faults. The main goal of functional safety verification is to avoid safety goal violations, not general failures in the design. This is the concept of safe faults. Our hypothesis is on deploying the strengths of existing technologies in an innovative methodology to resolve the issues. As a result, this chapter proposes a novel methodology based on the innovative use of existing technologies that address the problem. The main contributions can be listed and summarized as follows:

- A new systematic approach combined with engineering concepts in order to deliver an industrial solution that can be deployed for SoC targeting the automotive industry.
- An automated safe fault identification technique supported by an industrial-grade electronic design automation (EDA) tool flow: logic simulation of the target design when it runs the software application, extraction of coverage reports that reflects the behavior of the software application, development of formal properties that are translated from coverage reports, and formal analysis execution.
- ISO 26262-driven safe fault identification technique that contributes to the testing and verification theory by focusing the test efforts on the other faults (dangerous).
- A scalable formal property generation approach to translate the design's operational behavior into the formal analysis tool.
- An experimental demonstration of the effectiveness of the proposed technique on a comprehensive automotive benchmark SoC, using its CPU and the UART and CAN peripherals.
- Significant improvements in the classification of safe faults and of the resulting DC, thus allowing to achieve a higher safety level. When the AutoSoC runs the CCA, 20%, 11%, and 13% of all faults in the CPU, UART, and CAN are classified as safe using the presented technique, respectively. The value of DC is increased by around 6% and 4% for the CPU and the CAN, respectively. This analysis also reduces the number of undetected faults by 1.5 and 1.6 times in the CPU and CAN, respectively.

6.2 Related Works

Many works exist in the literature about hardware fault classification. This sub-chapter examines some of them based on different approaches, such as simulation-based fault injection, formal methods, ATPG, or hybrid approaches.

Several works have explored simulation-based fault injection targeting fault classification. For example, [31] optimizes fault injection campaigns by integrating it into the design verification environment and using the clustering approach to accelerate the campaigns. However, using only simulation-based fault injection for fault classification is computationally expensive and incomplete; hence it requires additional methods to classify undetected faults. Similarly, [94] relies only on simulation-based fault injection to classify the faults, but there was no additional classification technique proposed. Similarly, [95], [96], and [97] deploy simulation-based fault injection to classify faults in automotive systems considering the requirements of ISO 26262. In short, when simulation-based fault injection is used alone to classify faults, additional techniques targeting the classification of undetected faults are necessary.

Hence, some other works have investigated formal analysis, focusing on safe fault identification. [98], [99], and [100] use the ability of the formal techniques to analyze the design behavior. Safe fault identification is also applied to GPUs. For example, [101] employs formal analysis to increase fault coverage when the identification technique is applied to an open-source GPU. These works specifically focused on identifying structurally safe faults, i.e., faults for which there are no test or input stimuli due to the hardware structure, independently of the software and the application.

Researchers have also combined simulation-based fault injection and formal analysis leveraging fault classification. [81] and [38] have an eclectic approach that makes use of the strength of different technologies. Even though these works are promising in terms of the results, they still require many manual efforts based on the engineer's expertise.

On the other side, ATPG is also a promising technique to identify safe faults. Examples of this approach are [102], [103], and [104], which aim at identifying untestable faults in sequential circuits. We note that untestable faults are, by definition, safe faults [105]. In addition, [106], [105], and [93] resort to ATPG to identify application-dependent safe faults, which is the same target of the work described in this paper. Even though these works can identify safe faults using the ATPG, they still have a manual part in their flow, i.e., they are semi-automated.

Considering application-dependent safe faults, some works have proposed solutions for the classification of these kinds of faults. For example, [107] explores the use of safe faults to optimize STL fault coverage in microprocessors, which is not safety critical. However, the scope of the work is limited only to CPU modules, and the deployed tests are not automotive representative.

To address the outlined gaps, the technique proposed in this chapter corresponds to a fully automated fault classification technique, which focuses on safe faults when a CPU is running a specific SW application. The main strength of the proposed approach lies in the developed formal properties, which are extracted via the analysis of the target system's operational behavior.

6.3 Background

This sub-chapter, first, provides basics about hardware fault classification. Then, simulation-based fault injection and formal analysis are explained in the context of the hardware fault classification.

6.3.1 Hardware Fault Classification

ISO 26262 divides the malfunction of electrical/electronic components into two categories, corresponding to systematic and random faults [12]. A systematic fault is manifested in a deterministic way and can only be prevented by applying process or design measures. On the other hand, a random fault can occur unpredictably during the lifetime of a hardware element. When safety-critical designs are considered such as automotive, medical, or aerospace, safety and verification engineers must prove that both the correct and safe functionality of these designs are guaranteed, taking into account both systematic and random faults. In this chapter, we focus on random faults, only.

Several sources exist for random hardware faults, such as extreme operating conditions, aging, or in-field radiation. Also, each fault type should have a fault model that describes how faults from these sources should be modeled at the appropriate hardware design abstraction level (e.g., at the gate or RT level). Moreover, faults can be permanent and transient. Transient faults occur and subsequently disappear. On the other hand, permanent faults occur and stay until removed or repaired. This chapter focuses on permanent faults modeled as stuck-at faults, i.e., signals getting permanently stuck at a given logic value, i.e., 0 (stuck-at-0, SA0) or 1 (stuck-at-1, SA1), following what safety standards in the automotive domain suggest. We also note that a stuck-at fault can apply to all netlist signals, such as ports of logic gates or registers.

In order to determine the probability of a fault causing a safety-critical failure, its effects must be classified into two different categories as follows.

- *Safe*: A safe fault does not disturb any safety-critical functionality because it is not in safety-relevant logic or is in safety-relevant logic but is unable to impact the safety-critical functionality of a design (i.e., it cannot violate a safety goal).
- *Dangerous*: A dangerous fault impacts the safety of the device and creates a hazard that may produce a safety goal violation.

6.3.2 Simulation-based fault injection

As an integral part of the safety-critical IC development, simulation-based fault injection is a widely-used technique to identify fault effects [108]. simulation-based fault injection tools analyze an RTL or gate-level abstraction of an IC by performing a simulation with some given test stimuli. In general, the fault injection flow is based on the comparison between the results of the *Good Run* and those of the *Faulty Run*. First, the *Good Run* is run to generate reference values. In this step, observation points where the propagation of faults is monitored are specified. Then, the *Faulty Run* is executed with faults injected. In the end, the reference values obtained by the *Good Run* and the faulty values generated by the *Faulty Run* are compared for the classification of each injected fault and we can determine whether each injected fault is *detected* or *undetected*. Faults are classified as detected when at least one output value changes for a specified observation point between the *Good Run* and the *Faulty Run*. Otherwise, the fault is classified as undetected.

Although simulation-based fault injection is a widely used and adopted technique by both industry and academia, it suffers from two problems [41]:

- *Incomplete results*: It is impossible to simulate all possible combinations of input sequences when considering today's complex applications and devices. Hence, some faults cannot be accurately classified as Safe or Dangerous with the simulation-based fault injection technique.
- *No-effect faults*: Faults injected into components of the target system that will not be activated during the execution of a workload (testbench) will result in no-effect faults. These faults are classified as undetected by the simulation-based fault injection. This causes ambiguous results because these faults might be Dangerous when different or more comprehensive input stimuli are used.

Because of the two reasons listed above, it may be required to use additional classification techniques, such as formal methods as explained in the following subsection, to classify faults after simulation-based fault injection, whether they are safe or not. We must also mention that both sets of detected and undetected faults may contain safe and dangerous faults, as illustrated in Fig. 28. Therefore, if a fault is not classified and not proven safe, it should be pessimistically considered as dangerous.

6.3.3 Formal Methods

Formal methods help to classify faults based on their effects. An analysis is performed to determine whether or not a target design satisfies a set of properties or conditions. This approach is usually a combination of different techniques that employ static analysis and algorithmic calculations. Compared to simulation-based fault injection that applies one

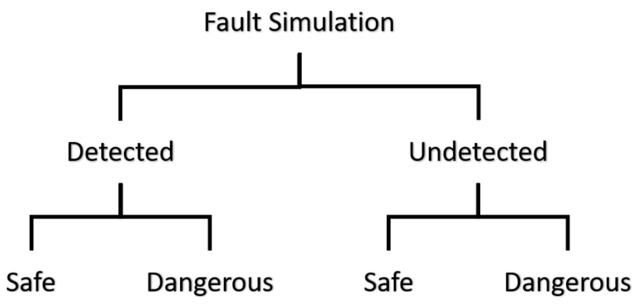


Figure 28: Fault Classification [10]

single stimulus, formal analysis is less limited since it abstracts from any specific stimulus. On the other hand, the computational complexity may limit the formal analysis applicability [8]. In this case, classification of all faults can be impossible; thus, a formal analysis tool should be fed by formal properties, developed carefully considering the constraints from the SW application, and looking for a compromise between computational feasibility and result accuracy, as it is done in this work.

In general, formal tools apply two checks, *Structural* Analysis Check and *Formal* Analysis Check to identify safe faults, as explained below.

6.3.3.1 Structural Analysis Check Types

In the Structural Analysis Check, formal tools use the topological characteristics of a design to determine the testability of each fault. There are three methods of Structural Fault Analysis:

- Out-of-Cone of Influence (COI) Analysis: This method checks whether a given node is outside the COI of a given observation point(s); in that case, the fault is safe. In Fig. 29, all faults located on nodes in the COI of out_1 (shown in green) are safe since the considered observation point is out_0 in the example analysis. It is obvious that stuck-at faults on the cell ports of G3 cannot propagate to out_0 as they have no physical connection with out_0. Hence, faults on G3 are safe.

- Unactivatable Analysis: this is to check if a SA0 or SA1 fault is located on a node that is constant 0 or 1; if so, the fault cannot be activated. In this case, the fault is unactivatable and safe. In Fig. 30, assuming that in_0 is tied to logic zero, f_0 for SA0 is unactivatable and safe.

- Unpropagatable Analysis: this is performed to check if a fault is activated and in the COI of the considered observation point but cannot be propagated to the outputs. In this case, the fault is safe. In Fig 30, the AND gate G2 can block the propagation of f_1 if one of the in_1 or in_2 is always set with the logic value zero. Hence, f_1 would be safe for SA1 or SA0 as it can never be propagated to out_0.

6.3.3.2 Formal Analysis Check Types

As opposed to Structural Analysis Checks for which physical connections of a design are taken into account, Formal Analysis Checks are used to classify faults as well. The approach uses Good Machine and Bad Machine similar to the simulation-based fault injection and

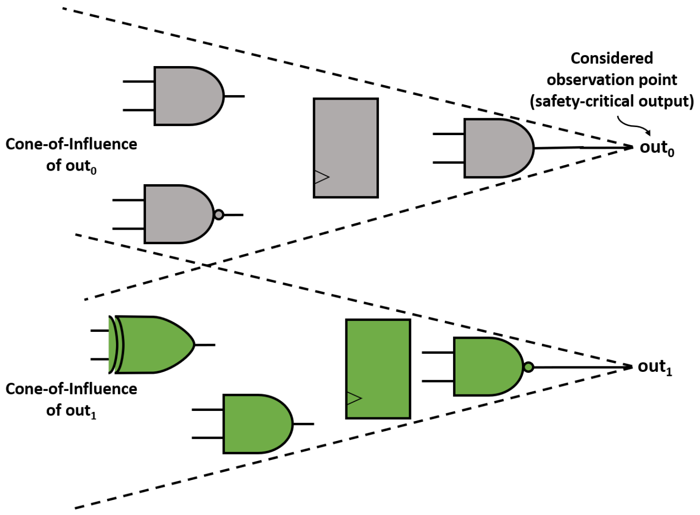

Considered observation point (safety-critical output)
out_0
Cone-of-Influence of out_0
Cone-of-Influence of out_1
out_1

Figure 29: Out-of-COI Example [10]

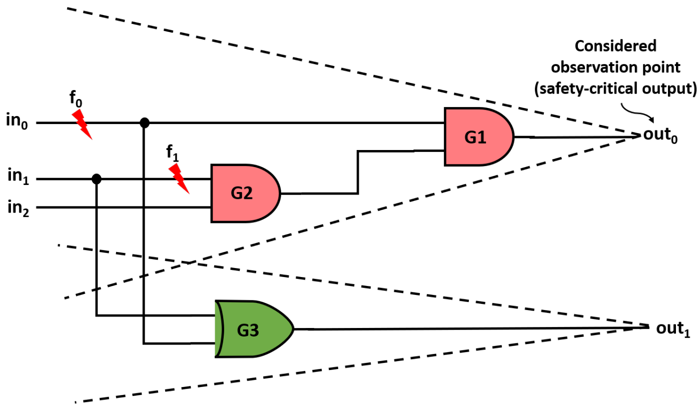


Figure 30: Unactivatable and Unpropagatable Analysis Example [10]

injects a fault in Bad Machine for formal analysis. In the end, the output signal values of Good and Bad Machines are compared to check whether an injected fault is propagated or not. A formal tool generally generates a Boolean representation of the function implemented by the circuit (or part of it) and uses formal techniques as explained above to prove this Boolean equation. Formal analysis tools use various engines based on Boolean expressions representation and manipulation techniques, such as Binary Decision Diagrams (BDDs) [109] to prove the formal properties exhaustively. There are two types of this analysis:

- Activation Analysis: This analysis checks whether the fault can be functionally activated from the inputs. If not, then it is determined to be safe.
- Propagation Analysis: This one checks whether the fault can propagate to the relevant output(s). If it cannot, then it is safe.

The technique described in the following sub-chapter deploys both structural and formal analysis checks resorting to formal methods.

6.4 Proposed Application-Dependent Safe Fault Identification Method

In this sub-chapter, first we explain the definition and details of application-dependent safe faults. Then, we describe each step of the proposed technique.

6.4.1 Application-dependent Safe Faults

In Chapter 6.3, we explained that a safe fault does not disturb any safety-critical functionality because it is not located in any safety-relevant logic or is in a safety-relevant component. Based on this explanation, we further classify safe faults as follows:

- Structurally-safe (Str-Safe): These are faults that cannot be activated or propagated to the outputs of interest by any test sequence because of the design's structural constraints. For example, a fault in the redundant logic or a floating net (i.e., any net that does not have a load) is Str-Safe. Another example is supply0 and supply1 nets. Specifically, a SA0 fault on supply0 net and a SA1 fault on a supply1 net are Str-Safe. Finally, a SA1 fault on a pull-up gate and a SA0 fault on a pull-down gate are Str-Safe.

- Functionally-safe: As opposed to Structurally-safe faults, a test or test sequence for Functionally-safe faults exists, and their effects may propagate to design outputs. However, they do not affect any safety-critical functionality. For example, faults in the debug unit of a CPU not used due to hardware configuration are Functionally-safe.

The present chapter focuses on a subset of *Functionally-safe faults*, corresponding to *Application-dependent safe faults* (App-Safe). App-Safe faults are related to the SW application that the target system executes, and they cannot disturb the safety-critical functionality in the operational mode. Therefore, it can be said that a fault can be App-Safe for one software application but may be dangerous for another software application.

More specifically, the target system considered in this work performs a single software application during the whole operational life. During the operation in the field, this application and its input data set do not access all the design parts; thus, inaccessible components generate App-Safe faults. For example, if the SW application does not use any multiplication operation, all resources related to the multiplication opcode become App-Safe faults. Therefore, opcodes of an SW application are a good indicator for App-Safe fault identification. Referring to the multiplication example again, when the SW application, which runs on the target design, does not include multiplication opcode, the SW application does not trigger multiplication hardware in the arithmetic logic unit (ALU), so faults on these components contribute to the App-Safe fault list. Another example of App-Safe faults can be found in the Design-for-Test modules of the design. The SW application does not use these hardware elements during the normal operation mode; hence, the corresponding faults are App-Safe.

In the following sub-chapter, we explain the proposed flow to identify App-Safe faults in an industrial-size SoC when an SW application is being run on it.

6.4.2 The Proposed Flow

In Fig. 31, the proposed flow to identify App-Safe faults is shown step by step. At the beginning of the flow, we have a Design Under Test (DUT) circuit (typically, an SoC) and a SW application running on it. First, we run several logic simulations with different representative input data sets. The goal of running logic simulations is to analyze the design's behaviour when it runs the SW application. Next, application-specific formal properties are developed to translate the design's operational behavior into the formal environment. Formal properties provide input to the formal analysis tool to identify App-Safe faults. Finally, the formal analysis tool is deployed, and safe faults are listed. In the following subsections, we discuss each step in detail.

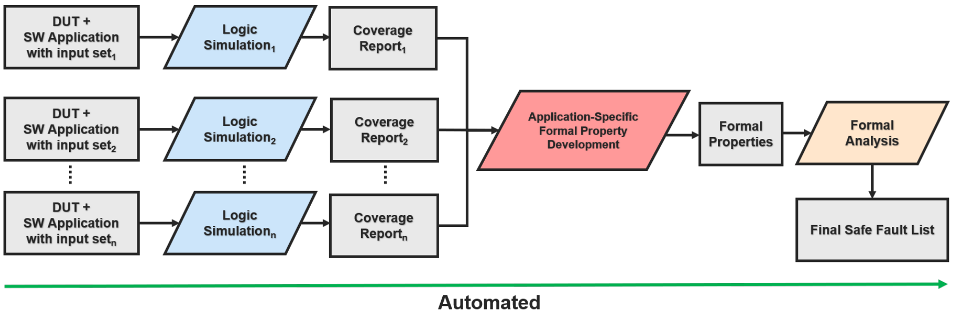


Figure 31: Proposed Application-dependent Safe Fault Identification Method [10]

6.4.2.1 Logic Simulation

In this step, we perform several logic simulations on the DUT executing the SW application with different representative realistic data input sequences, i.e. set$_1$ to set$_n$, as shown in Fig. 31. The aim of performing logic simulations is two-fold:

- to understand which design parts are affected by the input data set, and
- to extract the design's operational behavior when it runs an SW application.

To achieve the objectives, we generate hardware design code coverage data per each logic simulation and dump them into the coverage reports.

In general, logic simulations aim to detect which points are not toggled, as these are App-Safe candidates that must be addressed. Concerning coverage metrics, the proposed work focuses on hardware code coverage that assesses how well the stimuli exercise the design code by pointing to design components that did not meet the desired coverage criteria [110]. Our technique deploys toggle and block coverage sub-types of design code coverage to identify App-Safe faults. *Block Coverage* is a primary code coverage metric that identifies which lines in the code have been executed and which have not. On the other hand, *Toggle Coverage* monitors, collects, and reports the signal toggle activity, allowing the identification of unused signals or signals that remain at constant value 0 or 1.

The block and toggle coverage metrics provide insight into the SW application behavior during the operational life of an IC. Thus, we can identify App-Safe candidates included in the Functionally-safe fault list. More specifically, block coverage can indicate that some states are never activated, indicating that the SW application does not use the corresponding design components. Likewise, constant signals identified by toggle coverage can highlight invalid configurations, not utilized functions, among others. Moreover, the combination of block and toggle coverage data should be carefully analyzed because they can point out further information about the SW application's behavior. For the sake of an example, an untoggled signal may never activate a state machine block, and this can cause some other blocks to remain unactivated during the simulation. The small Verilog code in Listing 1 and Table 16 illustrates block coverage, toggle coverage and explains why both of them should be carefully analyzed. Listing 1 shows that rf_data_in block (at Line 6) is never activated since $break\_error$ is never toggled to logic 1 as shown in Table 16. This coverage results also means that rf_data_in never gets the right-hand side value at line 6 as the block is not activated. This example points out the importance of assessing block and toggle coverage together.

Listing 1: Block coverage example: rf_data_in is not executed [10]

```
 1  if(srx_pad_i | break_error)
 2  // The following "begin" block is covered (100%)
 3  begin
 4          if(break_error)
 5          // The following block is not covered (0%)
 6          rf_data_in <= {8'b0, 3'b100};
 7  else
 8          // The following block is covered (100%)
 9          rf_data_in <= {rshift, 1'b0, rparity_error,
10                                     rframing_error};
11          // The following block is covered (100%)
12          rf_push <= 1'b1;
13          rstate <= sr_idle;
14  end
```

Table 16: Toggle coverage example: break_error is not toggled [10]

Signal Name	0-to-1 Toggling	1-to-0 Toggling
break_error	0	0

6.4.2.2 Application-Specific Formal Property Development

The development cycle of ICs begins with inferring the specification and requirements of the target system. Also, the Design Under Test (DUT) must be verified with a formulated verification plan, which is defined by both Design and Verification engineers. Then, features or requirements of the DUT are created and mapped to the formal properties to deploy them in a formal analysis tool [111]. Formal properties are created from the design specification and implementation decisions. Thus, after extracting the target system's operational behavior through logic simulations, in this step, we translate this behavior to the formal properties to be used in a formal analysis tool, which will identify additional App-safe faults.

We use two types of formal properties to define the correct behaviour of the design. The first one is *assume statement*, which creates an assumption for the specified Boolean expression that evaluates to either true or false. In the general sense, it specifies that the given property is an assumption and is used to generate input stimulus. Hence, *assume statements* can be helpful when we define a design configuration or to inform the tool how the design inputs can behave. Without this assumption, a formal tool checks all possible input combinations of the DUT. There are two benefits of using *assume statements* in the formal environment. First, it allows excluding illegal input combinations when known. Legal inputs are those that we expect to see during normal operation. It is not realistic to expect the design to behave correctly when all possible input combinations are being applied unless we explicitly define every possible set of input combinations that the design can theoretically see. The second benefit of using *assume statements* is that it intentionally reduces the state space, which is exhaustive when no assumption is defined. For example, as we want to prepare our formal environment considering the design's operational behavior, we should disable the $scan_enabl$e pin as the scan chain is not activated during the operation and is used only for test purposes. In this case, the *assume statement* given in (9) is created to inform the formal tool about the $scan\_enable$ signal behavior; thus, the input test stimuli of a formal analysis tool are limited accordingly. (9) simply informs the tool that $scan\_enable$ is always logic-0. *Assume statements* also increases the safe fault identification capacity of a formal analysis tool by guiding it. Moreover, similar to the example given below, the input ports of design instances are suitable candidates for *assume statements*.

$$assume - env\,\{scan\_enable == 1'b0\} \tag{9}$$

The second formal property is *fault propagation barrier*, which creates a formal barrier that blocks the propagation of a fault. In this case, faults cannot propagate after this barrier; therefore, they cannot disturb any safety-critical functionality. For instance, knowing that the debug unit is not used in the design's operational mode, we can block all faults to propagate from it and identify more App-Safe faults. As seen in (10), the formal analysis tool is asked to block all faults propagated to du_dat_o, which is the debug unit's data output signal. As in this given example, output ports are proper candidates for a *fault propagation barrier* as opposed to *assume statements*, for which input ports are suitable candidates.

$$check\_fsv - barrier\,\{du\_dat\_o\} \tag{10}$$

Consequently, the application-specific formal properties [112] can be developed using *assume statements* and *fault propagation barriers*. By doing so, the internal architecture and logical details of the target system, the operational constraints (if any), or the initial configuration of the design can be defined as formal properties to be used in the formal analysis step. Therefore, the design's operational behavior can be transferred from the logic simulations into the formal analysis tool. The following sub-chapter explains how formal analysis tool uses these application-specific formal properties.

6.4.2.3 Formal Analysis

Having specified formal properties of a target design in a suitable notation, a formal analysis tool can be employed to generate App-Safe faults. The advantage of the formal analysis is that it provides a precise answer to whether a fault is propagated since it considers all possible input stimuli combinations (yet configured and limited thanks to *assume statements* as explained before) and hence eliminates the dependency on input stimuli. In this step of the flow, a formal analysis tool checks each fault in the target design to see whether it can be propagated to the observation points or not. If any input stimuli cannot propagate a fault, it is classified as safe; in our case, it is App-Safe. Otherwise, the fault falls into the dangerous category.

The formal analysis flow, which includes three phases, is shown in Fig. 32. Phase I begins with the creation of input files that are Formal Properties established in the previous stage and the DUT. Then, it continues with the development of the TCL setup script for the formal analysis tool. The setup script consists of Verilog files, libraries, and formal property files. The setup script first analyzes design and property files to check for syntax errors. Then, it defines clock and reset signals. The clock definition is to specify the characteristics of how the clock is driven during a formal analysis run. The reset specification aims at bringing the design to a known state and avoiding unreachable failure states. In the next step, warnings are generated by the formal analysis tool if there is a mismatch between formal properties and the DUT. For example, a signal tied to the ground in the DUT and the *assume statement* that defines this signal as if it is always logic-1 can create a mismatch, and a warning is generated. However, as we automatically translate coverage reports to the formal properties, this is not the case for the work proposed in this work. Then, in Phase II, the formal engine proves the formal properties by running the Structural and Formal checks as presented in Chapter 6.3.3. Finally, in Phase III, App-Safe faults are identified and reported.

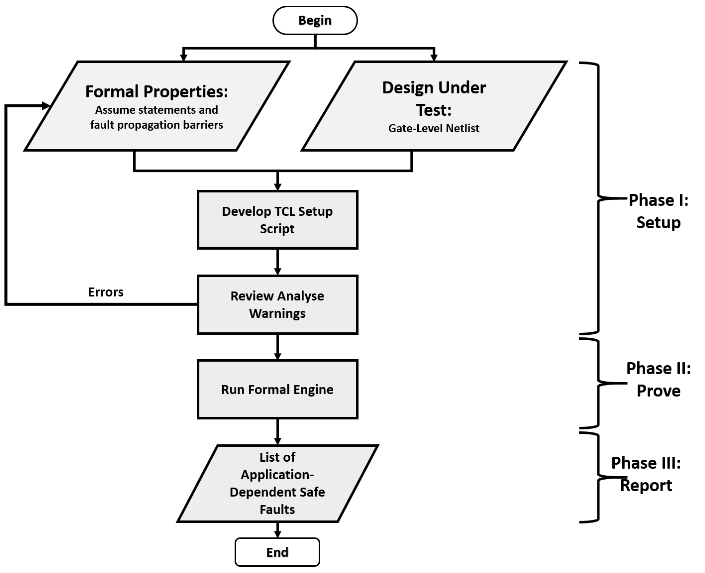


Figure 32: Formal Analysis Flow [10]

In brief, a formal analysis tool uses formal properties to generate safe faults. When we include formal properties driven by SW application, as mentioned before, we enable the tool to work in a well-specified configuration. Hence, formal analysis with the formal properties increases the number of identified App-Safe faults.

6.5 Experimental Setup and Results

The proposed application-dependent safe fault identification method is evaluated on the AutoSoC Benchmark suite, which was conceptualized in **VII**. All the details of the AutoSoC is presented in Chapter 4. This chapter first describes the experimental setup we used to quantitatively assess the effectiveness of the proposed approach. Then we provide the results in separate sub-chapters.

In order to demonstrate the effectiveness of the proposed App-Safe fault identification method, we used the experimental setup shown in Fig. 33. Our setup is composed of two AutoSoC nodes; each includes a CAN and a UART to communicate with each other, and one of the two AutoSoCs (named AutoSoC-0) is assumed to be active, whereas the other (named AutoSoC-1) is the passive node. Moreover, CCA accesses CAN or UART in both the AutoSoC-0 and the AutoSoC-1. Thus, each CCA comes in two modes, even though the executed steps are symmetric; the two AutoSoC nodes alternatively receive and send messages in the same configuration. Furthermore, even if it is changeable, AutoSoC-0 receives messages first while AutoSoC-1 transmits first in our experimental setup. Finally, the whole system is simulated at the gate-level.

Concerning the EDA tools, we used Cadence Xcelium™ for logic simulations, Cadence® Integrated Metrics Center (IMC) for coverage analysis, Cadence® JasperGold® Functional Safety Verification (FSV) App for formal analysis, and Cadence® Xcelium™ Fault Simulator (XFS) for the simulation-based fault injection. However, the approach proposed in this chapter remains applicable to other tool flows as well.

In brief, we first performed logic simulations using the hardware configuration described before, which runs the CCA SW application using different input data sets, as shown in Fig. 31. Then, coverage reports are generated and translated into application-specific formal properties that configure the formal analysis environment according to the SW application's behavior. Finally, the formal analysis tool is deployed to identify App-Safe faults.

We present results in two ways. First, safe fault identification is shown in CPU, UART, and CAN, respectively. Second, combined results (simulation-based fault injection + formal analysis) are reported for the CPU and CAN modules (this step does not include the analysis of UART).

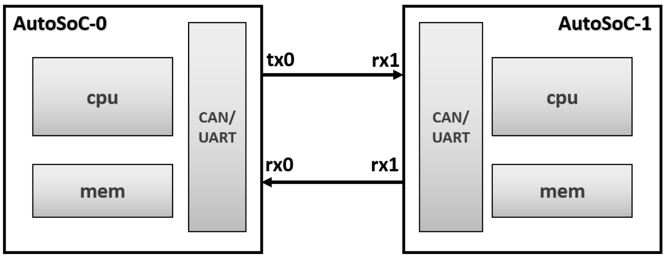


Figure 33: Experimental Setup [10]

6.5.1 Safe Faults in CPU

Firstly, App-Safe fault identification is checked in the CPU core (which has 96,354 faults in total) when it runs a SW application. We summarize the safe fault results of the CPU and safe faults with respect to total faults in the design (shown as %) in Table 17.

Table 17 categorizes the results based on the analysis we run. In the top row, it can be seen that we performed four analyses as follows:

- Application-independent: the Formal analysis tool is deployed on the gate-level netlist of the AutoSoC without any formal properties, meaning that the identified safe faults are valid for any SW application.
- Baremetal-CCA: the CCA runs bare-metal, which refers to running the SW application directly on a CPU without the support of an operating system. In order to perform this analysis, the gate-level netlist of AutoSoC and the formal properties (as explained in Chapter 6.4.2.2) are used as inputs to the formal analysis tool.
- RTEMS-CCA: unlike Baremetal CCA, the SW application runs on an operating system in this analysis, meaning that it can start and stop different processes concurrently. The RTEMS CCA causes higher signal activity when compared to Baremetal-CCA as it runs on operating systems that trigger more signals. Also, RTEMS-CCA uses two additional opcodes than Baremetal-CCA. This means that RTEMS-CCA triggers more design components than Baremetal-CCA.

- Baremetal-Sum: For this analysis, we use an entirely different SW application than CCA. The application performs a sum operation, and it has fewer opcodes than Baremetal-CCA. This SW application aims to show how App-Safe faults change when the CPU is running a different application.

In brief, App-Safe faults are originated from what a SW application executes in an IC. For example, some design components are not accessed during the design's operational life, such as debug units or scan chains. In addition, unused opcodes cause App-Safe faults, meaning that if (for example) the multiplication opcode is not used in the SW application that runs on the IC, all signals related to multiplication hardware become App-Safe faults as they are not exercised. Table 17 reports the results for the CPU core, also detailing the results achieved on each component module inside it. In the application-independent analysis that is shown in the second column, the formal analysis tool identifies 8.785% safe faults with respect to all faults in the CPU. We highlight that all the identified safe faults in the application-independent analysis are Str-Safe faults because the formal tool could not identify any safe faults using Formal Fault Analysis Check Types mentioned in Chapter 6.3.3.2 without formal properties. Concerning the three application-dependent analyses:

- the *Top* module includes connectivity signals and configuration-related signals. Among these, debug unit's address and data signals, interrupt request signals, multicore configuration signals, special-purpose-register signals are identified as safe in all analyses since they are not activated due to the SW applications configuration. De-pending on the opcodes used in the applications, there are slight differences in Baremetal-CCA, RTEMS-CCA, and Baremetal-Sum. For example, RTEMS-CCA trig-gers exception signals, which are connected to the top level.

- the *Decode_execute Unit* is the module where the instruction memory management unit (IMMU) and the data memory management unit (DMMU) signals take part. Many safe faults are identified in the IMMU and DMMU, which are not used by the SW applications. The number of safe faults is different between Baremetal-CCA and RTEMS-CCA because of the exception signals used by RTEMS-CCA, as mentioned above. In addition, the deviation between Baremetal-CCA and Baremetal-Sum is due to division and multiplication-related signals, which Baremetal-Sum does not use.

- the *Load-Store Unit* computes the addresses used by load and store instructions.

Table 17: Safe Faults in CPU [10]

	Application-ind.		**Baremetal-CCA**		**RTEMS-CCA**		**Baremetal-Sum**	
CPU Modules	**Safe Faults**	**%**	**Safe Faults**	**%**	**Safe Faults**	**%**	**Safe Faults**	**%**
top	1,679	1.743%	1,725	1.790%	1,716	1.781%	1,717	1.782%
register file	2	0.002%	5	0.005%	2	0.002%	5	0.005%
decode_execute unit	651	0.676%	844	0.876%	719	0.746%	949	0.985%
load store unit	910	0.944%	2,380	2.470%	2,317	2.405%	2,380	2.470%
writeback mux unit	0	0.000%	0	0.000%	0	0.000%	76	0.079%
fetch stage	976	1.013%	1,230	1.277%	1,195	1.240%	1,230	1.277%
control stage	3,966	4.116%	11,618	12.058%	6,418	6.661%	11,618	12.058%
arithmetic logic unit	55	0.057%	1,000	1.038%	1,000	1.038%	19,478	20.215%
decode unit	5	0.005%	267	0.277%	16	0.017%	315	0.327%
branch pred. unit	0	0.000%	0	0.000%	0	0.000%	0	0.000%
TOTAL	8,465	8.785%	19,484	20.221%	13,670	14,187%	38,193	39.638%

Safe faults may exist, as not all addresses are used by the SW applications. In addition, some connection signals create a slight difference between Baremetal-CCA and RTEMS-CCA.

- the *Fetch stage* fetches the next instruction from memory into the instruction register. Therefore, it is directly associated with the address range, which is not fully covered by the SW application. Therefore, safe faults can be identified in this unit. In addition, the difference between Baremetal-CCA and RTEMS-CCA is due to exception signals.

- the *Control stage* has the most considerable impact on the number of identified safe faults. This unit contains features such as tick-timer, interrupts, and configuration registers. Since the CPU configuration is the same in all applications, configuration registers create the same amount of safe faults. However, the tick-timer unit has a higher activity in RTEMS-CCA; hence, it has fewer safe faults when the CPU runs RTEMS-CCA.

- Concerning the *Arithmetic Logic Unit*, the proposed technique identifies the same amount of safe faults in Baremetal-CCA and RTEMS-CCA as they use the same arithmetic opcodes. However, Baremetal-Sum performs only addition operations; therefore, all the other arithmetic operations contribute to the safe faults.

- the *Decode Unit* is directly affected by the used opcodes; hence, there is a difference between the numbers of safe faults, as all three analyses use different numbers of opcodes.

The results in Table 17 show that the percentage of safe faults varies widely from one module to another, depending on the tasks performed by the modules. Also, the number of App-Safe faults is relevant, accounting for about 20%, 14%, and 40% in Baremetal-CCA, RTEMS-CCA, and Baremetal-Sum applications, respectively.

6.5.2 Safe Faults in UART

Concerning the UART module, which has 19,120 faults in total, we followed the same procedure using two scenarios (application-independent and CCA is compared), and the results are detailed in Table 18. We also noted that there is no difference between Baremetal-CCA or RTEMS-CCA, so we only report the identified safe faults as CCA in Table 18. In short, the proposed technique identified 11.088% safe faults, which is two times more when compared to the application-independent analysis.

More specifically;

- The *regs* unit has configuration registers, whose value is written in the initialization phase. Since the UART configuration is fixed in CCA, some parts of the UART are unused; thus, several safe faults can be identified in this unit.

- Safe faults in the *transmitter* module originate from the configuration of the transmission format, such as the selected BAUD rate. Therefore, more safe faults can be found in this unit when the SW application is fixed, as in this work. Correspondingly, *transmitter fifo* is partially affected by these factors.

- Concerning the *receiver* module that is directly affected by the configuration registers, a significant amount increase in the number of safe faults is observed. This mainly stems from the fact that the receiver module is responsible for generating interrupts. However, the CCA works in polling mode, meaning that no interrupt is used. Moreover, the receiver module has a Modem configuration, which CCA does not need. By extension, *receiver fifo* is partly affected, similar to transmitter fifo.

Table 18: Safe Faults in the UART [10]

UART Modules	**application-ind.**		**CCA**	
	Safe Faults	**%**	**Safe Faults**	**%**
top	9	0.047%	19	0.099%
wb_interface	78	0.408%	78	0.408%
regs	357	1.867%	1,003	5.246%
transmitter	67	0.350%	67	0.350%
uart_sync_flops	6	0.031%	6	0.031%
fifo_tx	101	0.528%	101	0.528%
receiver	171	0.894%	651	3.405%
fifo_rx	195	1.020%	195	1.020%
TOTAL	**984**	**5.146%**	**2,120**	**11.088%**

6.5.3 Safe Faults in CAN

The same analysis is performed for the CAN module, which has 38,012 faults in total, and the results are provided in Table 19. In the application-independent analysis, the formal analysis tool can classify only 1.415% of all faults as safe. On the other hand, when the proposed approach is deployed, the amount of safe faults is increased to 12.909%, which is not negligible.

Similar to UART, the number of safe faults in CAN is directly affected by its configuration. In CCA, we configure the CAN to work in peliCAN mode, which has extended frame format messages. When the basiCAN mode is used, more safe faults can be identified. To put the results given in Table 19 more explicitly;

- *Acceptance_code_mask* defines whether the corresponding incoming bit is compared to the respective bit in the *acceptance_code_regs*. Similarly, *bus_timing_regs* defines the values of the Baud Rate Prescaler, programs the period of the CAN system. Moreover, *clock_divider_regs* controls the clock frequency for the microcontroller and allows to deactivate the clock pin. In addition, the CCA works in polling mode, so safe faults can be found in the *IRQ* registers. Consequently, all these registers should not be changed after the initial configuration; thus, this creates additional safe faults.

- *Bit Timing Logic* is directly affected by *bus_timing_regs* explained above, so the CCA originates some safe faults in this unit.

- *Bit Stream Processor* corresponds to the control and processing unit of the peripheral. It is a sequencer that controls the data stream between the transmit buffer, the receive fifo, and the CAN bus. Also, error-detection, arbitration, stuffing, and error-handling are done in this unit. In addition, the Bit Stream Processor is affected by the configuration, such as working mode of the CAN like listen only mode or self test mode. The CCA does not use these modes, which provide safe faults shown in Table 19.

- *Acceptance filter* checks whether the message currently on the bus has to be stored by the peripheral or not. If the message is accepted, it is stored in the fifo. In other words, bit acceptance filter and its fifo is related to *acceptance_code_regs* and *acceptance_code_mask*; therefore, the fixed content of these registers gives rise to safe faults.

Table 19: Safe Faults in the CAN Controller [10]

CAN Modules	**application-ind.**		**CCA**	
	Safe Faults	**%**	**Safe Faults**	**%**
top	10	0.026%	41	0.108%
can_registers	22	0.058%	769	2.023%
acceptance_code_regs	0	0.000%	52	0.137%
acceptance_mask_regs	0	0.000%	52	0.137%
bus_timing_regs	0	0.000%	26	0.068%
clock_divider_regs	11	0.029%	41	0.108%
command_reg	13	0.034%	57	0.150%
error_warning_reg	10	0.026%	74	0.195%
irq_en_reg	0	0.000%	15	0.039%
mode_regs	14	0.037%	53	0.139%
tx_data_regs	0	0.000%	115	0.303%
bit timing logic	46	0.121%	299	0.787%
bit stream processor	354	0.931%	2,988	7.861%
can_crc_rx	0	0.000%	0	0.000%
acceptance filter	3	0.008%	256	0.673%
can_fifo	55	0.145%	69	0.182%
TOTAL	**538**	**1.415%**	**4,907**	**12.909%**

6.5.4 Combined Results: Simulation-based Fault Injection + Formal Analysis

In this step, we combine simulation-based fault injection and formal analysis, as it is proposed in this work, to check the increase in the DC. This analysis targets the CPU and the CAN modules in the AutoSoC.

As mentioned in Section 6.1, the simulation-based fault injection is not enough to classify all faults. It is needed to analyze the undetected faults to check if the desired DC is reached. If the DC does not match the requirements, then undetected faults must be re-analyzed using alternative methods, such as the proposed technique in this work. In short, the purpose of this step is to show that the proposed technique can increase DC to achieve the figures required by a given Automotive Safety Integrity Level.

In order to do this analysis, we resorted to the Software-Based Self-Test (SBST) [92] approach in the form of STLs. In the considered scenario the AutoSoC runs Baremetal-CCA in the field, and the STL, when activated, forces the processor to execute a proper sequence of instructions. Then, a signature is produced based on the generated results, and the application can compare it with the expected results if there are faults. The developed STL for the AutoSoC CPU is a combination of 57 test programs, partly taken from [7] and partly newly developed for this chapter. Concerning the STL for CAN, we use the same test programs described in [8]. The STL was developed as a collection of tasks that can either operate independently or collectively, depending on the self-test time slot [54].

The following steps are applied;

- First, Str-Safe faults are identified using Cadence® JasperGold® Functional Safety Verification (FSV) App.
- Second, we use the Cadence® Xcelium™ Fault Simulator to inject SA0 and SA1 faults at cell ports of the AutoSoC CPU and CAN modules, which run the STL as a workload. As a result, faults are classified as Detected or Undetected.
- Second, DC is calculated using (8).
- Third, App-Safe faults identified before excluded from Undetected faults. This process is incremental, always focusing on faults that were previously Undetected.
- Finally, DC is calculated again with the newly achieved numbers using (8).

Fig. 34 and Fig. 35 detail the results of the STL efficiency and uptrend in DC when App-Safe faults are identified. Concerning the analysis in CPU, Fig. 34 shows that 8,465 Str-Safe faults are identified in the beginning. Then, when simulation-based fault injection is deployed, 71,255 Detected and 16,634 Undetected faults are classified. After simulation-based fault injection, DC is 81.07%, calculated using (8). Then, by applying the proposed safe fault identification technique using formal methods, 5,627 App-Safe faults are identified, i.e., Undetected faults are reduced to 11,007. Using again (8), DC is increased to 86.62%. A similar analysis is performed in CAN as shown in Fig. 35. As a result, DC increased from 88.04% to 91.97%.

The proposed technique appears as a promising way for the classification of undetected faults via safe fault identification. Combined results show that DC is improved by around 6% for the CPU and 4% for the CAN. Moreover, with a final DC of 91.97%, the CAN achieves the requirements for an automotive ASIL B [12] hardware component as-is, i.e., without design modifications.

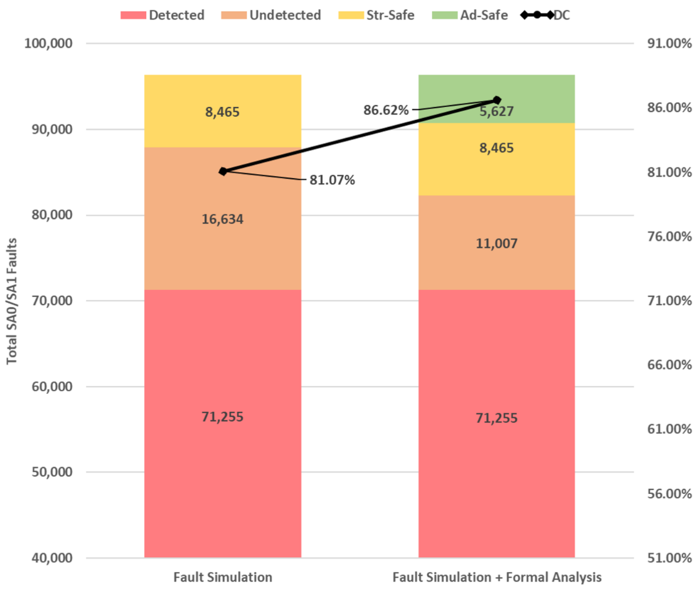


Figure 34: Combined Results in CPU [10]

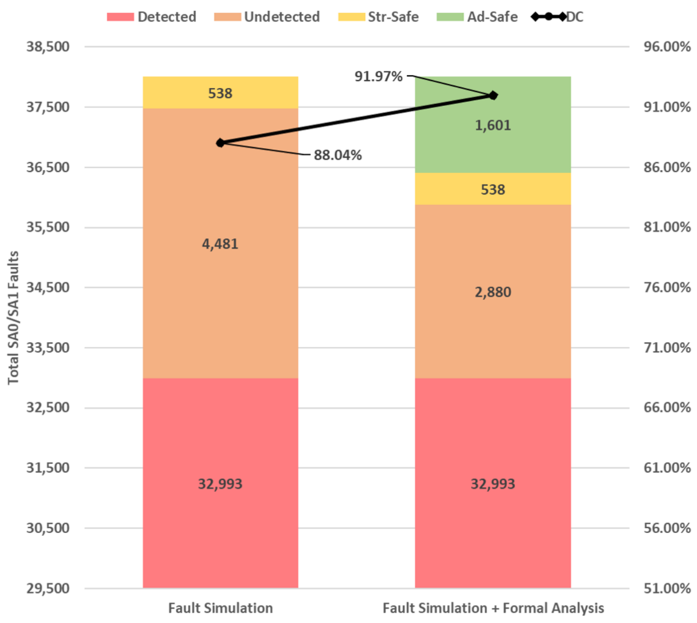


Figure 35: Combined Results in CAN [10]

6.6 Chapter Conclusions

Functional safety verification is a crucial and non-negotiable requirement that must be considered throughout the safety-critical IC design cycle. Therefore, the ISO 26262 functional safety standard has been developed to guide how this requirement is implemented. According to ISO 26262, random hardware failures can occur unpredictably during the lifecycle of an IC. Thus, random hardware faults must be classified based on their effects, whether they disrupt any safety-critical functionality or not. Nevertheless, this classification process is expensive and error-prone since it requires a combination of tools and input from experts based on their design knowledge. The method proposed in this chapter brings a solution to this challenge.

The proposed methodology focuses on identifying safe faults on an automotive SoC when it runs a single SW application. We extend Functionally-safe faults by the identification of Application-dependent safe faults. The flow relies on code coverage analysis through logic simulations and formal methods. The methodology starts with the analysis of code coverage to understand the target system's operational behavior. In other words, faults that do not disturb any safety-critical functionality are first identified through code coverage analysis. Then, code coverage results are translated to formal properties, then transferred to a formal analysis tool to constrain the environment to identify safe faults. The proposed methodology is demonstrated on the AutoSoC using its CPU, UART, and CAN when the cruise-control application runs.

We computed the number of identified safe faults (specifically focusing on stuck-at faults). In addition, we combined simulation-based fault injection and the proposed formal technique to show the increase in Diagnostic Coverage. As a result, the number of safe faults accounts for 20%, 11%, and 13% in the CPU, CAN, and UART modules, respectively. Concerning the Diagnostic Coverage, we show that it is increased by 6% and 4% in CPU and CAN modules, respectively. This analysis also proves that the number of undetected faults for the same STL is reduced by 1.5-1.6 times for the CPU and CAN, significantly increasing the Diagnostic Coverage for an industry-scaled SoC with a sample automotive application.

7 Conclusions

Safety becomes a fundamental requirement for automotive SoCs. It means that the SoC will perform the intended function correctly or fail in a safe (anticipated) manner. In the context of this PhD thesis, the functional safety of E/E systems has been explored from different perspectives. In particular, the main focus is the optimization of analysis techniques, along with their relationship. The presented approaches provide concise and well-structured usage of functional safety analysis techniques.

First, Chapter 3 presents a functional safety verification and validation methodology, combining three fault analysis techniques to increase the tool confidence level and achieve a higher safety level. Formal analysis, simulation-based fault injection, and ATPG tools are deployed, and more specifically, ATPG was utilized to support simulation-based fault injection. The purpose of this chapter is two-fold. The former is to compare tool outputs to increase the confidence level, as required by ISO 26262. The latter is to decrease the overall efforts of ISO 26262 compliance by increasing fault detection rates. The presented methodology compares the results of tool outputs to identify possible malfunctions and points the discrepancies to the designer. Moreover, this methodology enables the use of test vectors and test benches generated by the ATPG tool for the simulation of faults in the simulation-based fault injection tool and the use of a formal analysis tool for safe fault identification, which also reduces safety analysis efforts. The experimental results show high fault detection rates and comprehensive tool output reports, contributing to ISO 26262 metric achievement.

Second, Chapter 4 provides an automotive representative SoC named AutoSoC. The functional safety research requires a comprehensive platform to evaluate the quality of their proposed solutions to the community. It is developed as it is difficult to access representative automotive designs. The AutoSoC is developed considering its commercial counterparts with all requirements. It is implemented as customizable so that several configurations would be possible to support more detailed researches. It includes all essential safety components, including both hardware and software resources. In addition, an automotive representative software application is also provided. Chapter 4 explains the architecture of the AutoSoC and presents the functional safety analysis results in RTL and gate-level. In short, the AutoSoC benchmark suite allows the research community to contribute more to functional safety research and state-of-the-art by quantitatively evaluating their solutions' effectiveness.

Third, Chapter 5 proposes two fault list pruning techniques targeting simulation-based fault injection campaigns for transient faults. Even at RTL, Fault injection requires an excessively long simulation time that prevents detailed evaluation of hardware components with a significant number of injections. Therefore, it is necessary to reduce the effort to support these campaigns. Chapter 5 proposes two solutions. The former is dynamic slicing-based fault list pruning that applies dynamic slicing on the RTL code to pinpoint fault injection time-steps accurately and allows injection of only critical faults in these time windows. In this way, the dynamic slicing-based fault list pruning narrows down the fault space and provides reduced simulation time. As a result of the experiments on an industrial-sized CPU, an average 5-10% extra gain in simulation time for fault injection is achieved, which significantly improves the total chip validation costs, as this phase is the most time-consuming. The latter is a mapping-based fault list pruning technique that enables the representation of gate-level SET faults by multiple SEU faults at RTL. The purpose is to change the abstraction level from gate-level to RTL and reduce the complexity of expensive gate-level fault injection campaigns. Each SET fault is mapped to a flip-flop, considering their cone-of-influence. Also, formal analysis is deployed to perform a propagation analysis to identify SET faults that propagate to flip-flip inputs in all cases (independent from the workload). Then, flip-flop sets are created that represent each gate-level SET fault, and fault space is calculated. Experimental results prove that the fault space is pruned up to tens to hundreds of times.

Finally, Chapter 6 enhances hardware fault classification with the help of formal analysis. According to ISO 26262, potential faults in the automotive system must be detected and correctly governed when the SoC runs on the field to avoid life-critical situations. Therefore, it is necessary to classify each fault based on their effects, whether they are safe or dangerous. This is usually performed by expert judgment; however, this is expensive and error-prone. Thus, Chapter 6 proposes an automated hardware fault classification technique that increases the accuracy of the process. The presented technique adopts a practical scenario in which the AutoSoC, an automotive representative SoC, runs a single software application during its operation life. The technique starts with the extraction of the AutoSoC behavior when it runs a CCA. This is done by hardware code coverage analysis. Then, this behavior automatically translated into formal properties to be used in the formal tool for safe fault identification. This also constraints the formal environment according to the software application that is under the analysis. Additionally, the CCA uses UART and CAN to communicate with the external world, making the presented technique more comprehensive and realistic. As a result, the formal analysis identifies the safe faults, accounting for 20%, 11%, and 13% for the CPU, UART, and CAN, respectively. Furthermore, the simulation-based fault injection is fed by these results, and safe faults are excluded from the campaigns to check if the DC of the deployed STLs is increased. Consequently, 6% and 4% increase in DC is observed for CPU and CAN modules. This analysis proves that the technique presented in Chapter 6 successfully increases the DC and makes it easy to meet required safety levels.

As future work, more automation can be provided to be used to support both permanent and transient fault analysis campaigns. Also, the characterization of workloads (or test benches) can be targeted to identify non-critical time-steps more efficiently and reduce manual effort. This also contributes to narrowing down the search space of formal analysis and reducing the efforts of safe fault identification. In addition, cross-layer approaches can be adopted and studied more, enabling fault analysis at several abstraction levels.

List of Figures

List of Tables

References

[1] A. C. Bagbaba, F. Augusto da Silva, and C. Sauer. Improving the confidence level in functional safety simulation tools for iso 26262. In *2018 Design and Verification Conference (DVCON)*, 2018.

[2] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer. Use of formal methods for verification and optimization of fault lists in the scope of iso26262. In *2018 Design and Verification Conference (DVCON)*, 2018.

[3] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Efficient methodology for iso26262 functional safety verification. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 255–256, 2019.

[4] Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Efficient fault injection based on dynamic hdl slicing technique. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 52–53, 2019.

[5] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 129–1295, 2019.

[6] Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Accelerating transient fault injection campaigns by using dynamic hdl slicing. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019.

[7] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Annachiara Ruospo, Riccardo Mariani, Ghani Kanawati, Ernesto Sanchez, Matteo Sonza Reorda, Maksim Jenihhin, Said Hamdioui, and Christian Sauer. Special session: Autosoc - a suite of open-source automotive soc benchmarks. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–9, 2020.

[8] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Sandro Sartoni, Riccardo Cantoro, Matteo Sonza Reorda, Said Hamdioui, and Christian Sauer. Determined-safe faults identification: A step towards iso26262 hardware compliant designs. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020.

[9] Ahmet Cagri Bagbaba, Maksim Jenihhin, Raimund Ubar, and Christian Sauer. Representing gate-level set faults by multiple seu faults at rtl. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2020.

[10] Ahmet Cagri Bagbaba, Felipe Augusto da Silva, Matteo Sonza Reorda, Said Hamdioui, Maksim Jenihhin, and Christian Sauer. Automated identification of application-dependent safe faults in automotive systems-on-a-chips. *Electronics*, 11(3), 2022.

[11] Arslan Munir. Safety assessment and design of dependable cybercars: For today and the future. *IEEE Consumer Electronics Magazine*, 6(2):69–77, 2017.

[12] International Standardization Organization. *ISO 26262 Road Vehicles - Function Safety*. International Organization for Standardization, ISO 26262-1:2018(E) edition, 2018.

[13] E. Mollick. Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75, 2006.

[14] R. S. Oliveira, J. Semião, I. C. Teixeira, M. B. Santos, and J. P. Teixeira. On-line bist for performance failure prediction under aging effects in automotive safety-critical applications. In *2011 12th Latin American Test Workshop (LATW)*, pages 1–6, 2011.

[15] M. Barke, M. Kärgel, W. Lu, F. Salfelder, L. Hedrich, M. Olbrich, M. Radetzki, and U. Schlichtmann. Robustness validation of integrated circuits and systems. In *2012 4th Asia Symposium on Quality Electronic Design (ASQED)*, pages 145–154, 2012.

[16] Daisuke Kobayashi. Scaling trends of digital single-event effects: A survey of seu and set parameters and comparison with transistor performance. *IEEE Transactions on Nuclear Science*, 68(2):124–148, 2021.

[17] Lars Reger. 1.4 the road ahead for securely-connected cars. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 29–33, 2016.

[18] Alessandra Nardi and Antonino Armato. Functional safety methodologies for automotive applications. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, page 970–975. IEEE Press, 2017.

[19] Eishi Ibe, Ken-ichi Shimbo, Hitoshi Taniguchi, Tadanobu Toba, Koji Nishii, and Yoshio Taniguchi. Quantification and mitigation strategies of neutron induced soft-errors in cmos devices and components. In *2011 International Reliability Physics Symposium*, pages 3C.2.1–3C.2.8, 2011.

[20] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[21] F. Ferlini. *Methodology to Accelerate Diagnostic Coverage Assessment: MADC*. PhD thesis, Universidade Federal de Santa, 2016.

[22] B. A. Tabacaru. *On Fault-Effect Analysis at the Virtual-Prototype Abstraction Level*. PhD thesis, Technische Universitat Munchen, 2019.

[23] Heinz Gall. Functional safety iec 61508 / iec 61511 the impact to certification and the user. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 1027–1031, 2008.

[24] Alessandra Nardi, Samir Camdzic, Antonino Armato, and Francesco Lertora. Design-for-safety for automotive ic design: Challenges and opportunities. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2019.

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

[26] Schmid Tobias. Safety analysis for highly automated driving. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 154–157, 2018.

[27] Haissam Ziade, Rafic Ayoubi, and R. Velazco. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1:171–186, 01 2004.

[28] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[29] Alfredo Benso and Paolo Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. 01 2003.

[30] Steve Pateras and Ting-Pu Tai. Automotive semiconductor test. In *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4, 2017.

[31] Dan Alexandrescu, Adrian Evans, Maximilien Glorieux, and Issam 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)*, pages 145–150, 2017.

[32] S. Marchese and J. Grosse. Formal fault propagation analysis that scales to modern automotive socs. In *2017 Design and Verification Conference (DVCON)*, 2017.

[33] A. Traskov, T. Ehrenberg, and S. Loitz. Fault proof: Using formal techniques for safety verification and fault analysis. In *2016 Design and Verification Conference (DVCON)*, pages 27–32, 2016.

[34] S. Praveen, Siva Yellampalli, and Ashish 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)*, pages 45–48, 2014.

[35] S. Arekapudi, Fei Xin, Jinzheng 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*, pages 23–28, 2002.

[36] 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)*, pages 1–1, 2012.

[37] U. Krautz, M. Pflanz, C. Jacobi, H.W. Tast, K. Weber, and H.T. Vierhaus. Evaluating coverage of error detection logic for soft errors using formal methods. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 1–6, 2006.

[38] Alessandro Bernardini, Wolfgang Ecker, and Ulf Schlichtmann. Where formal verification can help in functional safety analysis. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[39] S. Marchese and J. Grosse. Formal fault propagation analysis that scales to modern automotive socs. In *2017 Design and Verification Conference (DVCON)*, 2017.

[40] A. Traskov, T. Ehrenberg, and S. Loitz. Fault proof: Using formal techniques for safety verification and fault analysis. In *2016 Design and Verification Conference (DVCON)*, 2016.

[41] Alfredo Benso and Stefano Di Carlo. The art of fault injection. *Control Engineering and Applied Informatics*, 13:9–18, 12 2011.

[42] C. Albrecht. Iwls 2005 benchmarks. Technical report, International Workshop on Logic Synthesis (IWLS), 2005.

[43] Jinho Han, Youngsu Kwon, Yong Cheol Peter Cho, and Hoi-Jun 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)*, pages 313–316, 2017.

[44] Ádria Barros de Oliveira, Gennaro Severino Rodrigues, and Fernanda Lima Kastensmidt. Analyzing lockstep dual-core arm cortex-a9 soft error mitigation in freertos applications. In *2017 30th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 84–89, 2017.

[45] Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. Evaluation of diverse compiling for software-fault detection. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 531–536, 2015.

[46] Gennaro S. Rodrigues, Felipe Rosa, Ádria Barros de Oliveira, Fernanda Lima Kastensmidt, Luciano Ost, and Ricardo Reis. Analyzing the impact of fault-tolerance methods in arm processors under soft errors running linux and parallelization apis. *IEEE Transactions on Nuclear Science*, 64(8):2196–2203, 2017.

[47] A. Evans, M. Nicolaidis, S. 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)*, pages 727–732, March 2013.

[48] Daniel Mueller-Gritschneder, Marc Greim, and Ulf Schlichtmann. Safety evaluation based on virtual prototypes: Fault injection with multi-level processor models. In *2016 International Symposium on Integrated Circuits (ISIC)*, pages 1–2, 2016.

[49] Bogdan-Andrei Tabacaru, Moomen Chaari, Wolfgang Ecker, Thomas Kruse, and Cristiano Novello. Fault-effect analysis on system-level hardware modeling using virtual prototypes. In *2016 Forum on Specification and Design Languages (FDL)*, pages 1–7, 2016.

[50] International Standardization Organization. *Road vehicles — Cybersecurity engineering*. International Organization for Standardization, ISO/SAE 21434:2021 edition, 2021.

[51] D. Lampret et al. *OpenRISC 1000 Architecture Manual*. opencores.org, revision 0 edition, December 2012.

[52] Hanxing Chen and Jun Tian. Research on the controller area network. In *2009 International Conference on Networking and Digital Society*, volume 2, pages 251–254, 2009.

[53] Philips Semiconductors. SJA1000, Stand-alone CAN Controller. `https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf`, 2000. Online; last accessed 15 November 2021.

[54] Riccardo Cantoro, Sandro Sartoni, and Matteo Sonza Reorda. In-field functional test of can bus controllers. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–6, 2020.

[55] Uart 16550 transceiver. `https://www.latticesemi.com`. Online, Last accessed: 28.11.2021.

[56] Ernesto 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)*, pages 1–1, 2018.

[57] Pasquale Davide Schiavone, Ernesto Sanchez, Annachiara Ruospo, Francesco Minervini, Florian Zaruba, Germain Haugou, and Luca 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)*, pages 43–48, 2018.

[58] A. Maheshwari, I. Koren, and N. Burleson. Techniques for transient fault sensitivity analysis and reduction in vlsi circuits. In *Proceedings 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 597–604, 2003.

[59] Kelin J. Kuhn, Martin D. Giles, David Becher, Pramod Kolar, Avner Kornfeld, Roza Kotlyar, Sean T. Ma, Atul Maheshwari, and Sivakumar Mudanai. Process technology variation. *IEEE Transactions on Electron Devices*, 58(8):2197–2208, 2011.

[60] Mojtaba Ebrahimi, Adrian Evans, Mehdi B. Tahoori, Razi Seyyedi, Enrico Costenaro, and Dan Alexandrescu. Comprehensive analysis of alpha and neutron particle-induced soft errors in an embedded processor at nanoscales. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, 2014.

[61] Mojtaba Ebrahimi, Adrian Evans, Mehdi B. Tahoori, Enrico Costenaro, Dan Alexandrescu, Vikas Chandra, and Razi Seyyedi. Comprehensive analysis of sequential and combinational soft errors in an embedded processor. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1586–1599, 2015.

[62] Fiesta++ : a software implemented fault injection tool for transient fault injection. `https://repositories.lib.utexas.edu/handle/2152/28159`, 2014. Accessed: 10.11.2021.

[63] Raimund Ubar, Jaan Raik, Heinrich Vierhaus, S. Misera, and Roberto Urban. *Fault Simulation and Fault Injection Technology Based on SystemC*, pages 268–293. 01 2011.

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

[65] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, pages 284–293, June 1998.

[66] D. T. Smith, B. W. Johnson, J. A. Profeta, and D. G. Bozzolo. A fault-list generation algorithm for the evaluation of system coverage. In *Annual Reliability and Maintainability Symposium 1995 Proceedings*, pages 425–432, Jan 1995.

[67] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez. New techniques for speeding-up fault-injection campaigns. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, pages 847–852, March 2002.

[68] J. Raik, U. Repinski, M. Jenihhin, and A. Chepurov. High-level decision diagram simulation for diagnosis and soft-error analysis. *Design and Test Technology for Dependable Systems-on-Chip*, pages 294–309, 2011.

[69] X. Iturbe, B. Venu, and E. Ozer. Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 91–96, Sept 2016.

[70] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra. Processor core profiling for seu effect analysis. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, March 2018.

[71] W. Mansour, R. Velazco, R. Ayoubi, H. Ziade, and W. El Falou. A method and an automated tool to perform set fault-injection on hdl-based designs. In *2013 25th International Conference on Microelectronics (ICM)*, pages 1–4, Dec 2013.

[72] T. Bonnoit, A. Coelho, N. Zergainoh, and R. Velazco. Seu impact in processor's control-unit: Preliminary results obtained for leon3 soft-core. In *2017 18th IEEE Latin American Test Symposium (LATS)*, pages 1–4, March 2017.

[73] M. Jenihhin, A. Tšepurov, V. Tihhomirov, J. Raik, H. Hantson, R. Ubar, G. Bartsch, J. H. M. Escobar, and H. Wuttke. Automated design error localization in rtl designs. *IEEE Design Test*, 31(1):83–92, Feb 2014.

[74] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. Di Guglielmo, G. Pravadelli, and F. Fummi. Combining dynamic slicing and mutation operators for esl correction. In *2012 17th IEEE European Test Symposium (ETS)*, pages 1–6, May 2012.

[75] A. Papadimitriou, D. Hély, V. Beroulle, P. Maistri, and R. Leveugle. A multiple fault injection methodology based on cone partitioning towards rtl modeling of laser attacks. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[76] P. Vanhauwaert, P. Maistri, R. Leveugle, A. Papadimitriou, D. Hély, and V. Beroulle. On error models for rtl security evaluations. In *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, May 2014.

[77] D. Rossi, M. Omana, F. Toma, and C. Metra. Multiple transient faults in logic: an issue for next generation ics? In *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, pages 352–360, Oct 2005.

[78] N. Miskov-Zivanov and D. Marculescu. Mars-c: modeling and reduction of soft errors in combinational circuits. In *2006 43rd ACM/IEEE Design Automation Conference*, pages 767–772, July 2006.

[79] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(10):1614–1627, Oct 2010.

[80] A. Mochizuki, N. Onizawa, A. Tamakoshi, and T. Hanyu. Multiple-event-transient soft-error gate-level simulator for harsh radiation environments. In *TENCON 2015 - 2015 IEEE Region 10 Conference*, pages 1–6, Nov 2015.

[81] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 129–1295, 2019.

[82] Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Efficient methodology for iso26262 functional safety verification. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 255–256, 2019.

[83] Edmund Clarke, Masahiro Fujita, Sreeranga Rajan, Thomas Reps, Subash Shankar, and Tim Teitelbaum. Program slicing for vhdl. *STTT*, 4:125–137, 10 2002.

[84] Mizuho Iwaihara, Masaya Nomura, Shigeru Ichinose, and Hiroto Yasuura. Program slicing on vhdl descriptions and its applications, 1996.

[85] B. Korel and J. Rilling. Dynamic program slicing in understanding of program execution. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*, pages 80–89, 1997.

[86] Opencores. `http://opencores.org`. Online, Last accessed: 10.11.2021.

[87] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, 2009.

[88] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Modular fault injector for multiple fault dependability and security evaluations. In *2011 14th Euromicro Conference on Digital System Design*, pages 550–557, Aug 2011.

[89] Gianpiero Cabodi and Marco Murciano. Bdd-based hardware verification. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification*, pages 78–107, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[90] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, Feb 1997.

[91] F. Corno, M. S. Reorda, and G. Squillero. Rt-level itc'99 benchmarks and first atpg results. *IEEE Design Test of Computers*, 17(3):44–53, July 2000.

[92] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design Test of Computers*, 27(3):4–19, 2010.

[93] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, and M. Sonza Reorda. About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, 2018.

[94] Kuen-Long Lu, Yung-Yuan Chen, and Li-Ren Huang. Fmeda-based fault injection and data analysis in compliance with iso-26262. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 275–278, 2018.

[95] Garazi Juez, Estibaliz Amparan, Ray Lattarulo, Joshue Perez Rastelli, Alejandra Ruiz, and Huascar Espinoza. Safety assessment of automated vehicle functions by simulation-based fault injection. In *2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*, pages 214–219, 2017.

[96] Yuting Fu, Andrei Terechko, Tjerk Bijlsma, Pieter J. L. Cuijpers, Jeroen Redegeld, and Ali Osman Örs. A retargetable fault injection framework for safety validation of autonomous vehicles. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 69–76, 2019.

[97] Frederico Ferlini, Laio Oriel Seman, and Eduardo Augusto Bezerra. Enabling iso 26262 compliance with accelerated diagnostic coverage assessment. *Electronics*, 9(5), 2020.

[98] Jaan Raik, Hideo Fujiwara, Raimund Ubar, and Anna Krivenko. Untestable fault identification in sequential circuits using model-checking. In *2008 17th Asian Test Symposium*, pages 21–26, 2008.

[99] M. Syal and M.S. Hsiao. New techniques for untestable fault identification in sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1117–1131, 2006.

[100] Hsing-Chung Liang, Chung Len Lee, and J.E. Chen. Identifying untestable faults in sequential circuits. *IEEE Design Test of Computers*, 12(3):14–23, 1995.

[101] Josie E. Rodriguez Condia, Felipe A. Da Silva, S. Hamdioui, C. Sauer, and M. Sonza Reorda. Untestable faults identification in gpgpus for safety-critical applications. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 570–573, 2019.

[102] Wei-Cheng Lai, A. Krstic, and Kwang-Ting Cheng. Functionally testable path delay faults on a microprocessor. *IEEE Design Test of Computers*, 17(4):6–14, 2000.

[103] Daniel Tille and Rolf Drechsler. A fast untestability proof for sat-based atpg. In *Proceedings of the 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems*, DDECS '09, page 38–43, USA, 2009. IEEE Computer Society.

[104] David E. Long, Mahesh A. Iyer, and Miron Abramovici. Fill and funi: Algorithms to identify illegal states and sequentially untestable faults. *ACM Trans. Des. Autom. Electron. Syst.*, 5(3):631–657, July 2000.

[105] C. Gursoy, M. Jenihhin, A. S. Oyeniran, D. Piumatti, J. Raik, M. Sonza Reorda, and R. Ubar. New categories of safe faults in a processor-based embedded system. In *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–4, 2019.

[106] Riccardo Cantoro, Sara Carbonara, Andrea Floridia, Ernesto Sanchez, Matteo Sonza Reorda, and Jan-Gerd Mess. Improved test solutions for cots-based systems in space applications. In Nicola Bombieri, Graziano Pravadelli, Masahiro Fujita, Todd Austin, and Ricardo Reis, editors, *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*, pages 187–206, Cham, 2019. Springer International Publishing.

[107] A. Narang, B. Venu, S. Khursheed, and P. Harrod. An exploration of microprocessor self-test optimisation based on safe faults. In *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Los Alamitos, CA, USA, oct 2021. IEEE Computer Society.

[108] Petra R. Maier, Uzair Sharif, Daniel Mueller-Gritschneder, and Ulf Schlichtmann. Efficient fault injection for embedded systems: As fast as possible but as accurate as necessary. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 119–122, 2018.

[109] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 419–422, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[110] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, 2001.

[111] Keerthikumara Devarajegowda, Lorenzo Servadei, Zhao Han, Michael Werner, and Wolfgang Ecker. Formal verification methodology in an industrial setup. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pages 610–614, 2019.

[112] J. Lach, S. Bingham, C. Elks, T. Lenhart, Thuy Nguyen, and P. Salaun. Accessible formal verification for safety-critical hardware design. In *RAMS '06. Annual Reliability and Maintainability Symposium, 2006.*, pages 29–32, 2006.

Acknowledgements

This thesis came to life thanks to the support of many people. I would like to express my sincere gratitude to everybody who has directly or indirectly contributed to my PhD studies.

In particular, I would like to thank Prof. Maksim Jenihhin, who always supported me, took the time, and guided me through my PhD studies. I would also like to thank Dr. Christian Sauer for giving me all kinds of opportunities and wise advice when it was needed. At the same time, I would like to express my gratitude to the team at Cadence Design Systems, who have always supported me when I need, and been very productive to work with. I would like to acknowledge the tremendous support of my beloved wife Ayşe, who always believed in me. I also wish to thank my family for always being with me and making their encouragement felt even from miles away.

Last but not least, I would like to acknowledge the organization that has supported my PhD studies: RESCUE European Training Network (ETN) project that received funding from European Union’s Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

Abstract
Methods to Optimize Functional Safety Assessment for Automotive Integrated Circuits

In recent years, the usage areas of electronics have been increasing rapidly. In parallel with this, due to technological developments, the complexity of these electronics increases, and they become more susceptible to errors. In addition, electronic devices must be designed and verified according to the area in which they are used and consider the requirements of this area. Especially if we consider areas such as aviation or automotive where safety is critical, we can easily deduce that even the slightest problem in electronics can threaten human life. Therefore, electronics used in safety-critical areas (typically SoC) must be designed and verified by following specific standards. Accordingly, in 2011, the ISO 26262 functional safety standard was developed for electronics to be used in automotive. ISO 26262 was revised in 2018.

According to this standard, SoCs used in automotive must contain safety mechanisms to prevent possible faults and their effects. Therefore, it should be determined how effective these safety mechanisms are in fault prevention, and the ASIL of the SoC or hardware should be determined according to the result. Various analysis methods can be used to do this, including those recommended by ISO 26262. Because, considering the hardware complexity of SoCs used in automotive today and the complexity of the software running on them, these analyzes can take much time. This increases the design costs of SoCs and even makes it difficult to meet the time-to-market criterion. For this reason, it should be carefully studied how these analysis methods can be made more efficient and optimal.

This PhD thesis focuses on making the above-mentioned functional analysis methods more efficient. Various methods have been proposed to do this, and their effectiveness has been proven by experimental studies and published. First of all, three tools that can be used in fault analysis were combined, and their results were compared in accordance with ISO 26262. At the same time, test vectors and test benches produced by ATPG were used instead of functional testbench in simulation-based fault injection tool in order to use the power of ATPG in detecting faults. Thus, the fault detection rate has been increased. Later, AutoSoC was proposed as a lack of open-source and representative SoC was identified in the field of functional safety. AutoSoC, as an openRisc-based, open-source, modular SoC containing safety mechanisms also recommended by ISO 26262, is a comprehensive platform where researchers can try the method they have developed. AutoSoC was presented as RTL and gate-level, some functional safety analyses were performed, and the results were shared. Next, simulation-based fault injection campaigns were optimized through fault list pruning: *(i)* The target designs were simulated, coverage results were analyzed, and critical time-steps and; therefore, critical faults were detected, then the fault injection campaign was carried out to inject only these faults. Thus, the execution times of fault injection campaigns were reduced by not injecting non-critical faults. *(ii)* By changing the abstraction level, fault injection was carried out only on these flip-flops by making simultaneous multiple fault injections. In this way, simulation-based fault injection campaigns were optimized by reducing the fault space. Additionally, safe fault identification has been performed in an automotive SoC (the AutoSoC) when it runs a CCA application during its operational life. In this work, which simulates a realistic situation, the parts that the CCA does not use in the circuit were determined through hardware code coverage analysis. In other words, the behavior of CCA was identified, and then this behavior was automatically translated into formal properties. Finally, the increasing trend in DC with the help of obtained safe faults was shown on the AutoSoC's CPU and CAN.

As a result, this PhD thesis has optimized and made the analysis methods required for automotive SoCs more efficient. This PhD thesis has a lot of promise for research in functional safety and safety-critical design.

Kokkuvõte
Meetodid autotööstuse kiipide funktsionaalse ohutuse hindamise optimeerimiseks

Viimastel aastatel on elektroonika kasutusalad kiiresti kasvanud. Paralleelselt sellega suureneb tänu tehnoloogilisele arengule elektroonikaseadmete keerukus ja need muutuvad vastuvõtlikumaks vigadele. Lisaks peavad elektroonikaseadmed olema projekteeritud ja kontrollitud vastavalt nende kasutusvaldkonnale ning arvestama selle valdkonna nõudeid. Eriti kui arvestada valdkondi, nagu lennundus või autotööstus, kus ohutus on kriitiline, võime järeldada, et isegi väikseim elektroonikaprobleem võib ohustada inimelu. Seetõttu tuleb ohutuskriitilistes piirkondades (tavaliselt kiipsüsteem) kasutatav elektroonika projekteerida ja kontrollida vastavaid standardeid järgides. Sellest lähtuvalt töötati 2011. aastal välja funktsionaalse ohutuse standard ISO 26262 autotööstuses kasutatava elektroonika jaoks.

Nimetatud standardi kohaselt peavad autotööstuses kasutatavad kiipsüsteemid sisaldama ohutus- mehhanisme, et vältida võimalikke rikkeid ja nende tagajärgi. Seetõttu on oluline kindlaks teha, kui tõhusad on need ohutusmehhanismid vigade ennetamisel, ja vastavalt tulemusele määrata kiipsüsteemi või riistvara mooduli ASIL tase. Selleks saab kasutada erinevaid analüüsimeetodeid, mida soovitab ka ISO 26262. Arvestades tänapäeval autotööstuses kasutatavate kiipsüsteemide riistvara ja nendel töötava tarkvara keerukust, võivad need analüüsid võtta palju aega. See suurendab kiipsüsteemide projekteerimiskulusid ja muudab turule jõudmise aja kriteeriumi täitmise keeruliseks. Sel põhjusel tuleks hoolikalt uurida, kuidas neid analüüsimeetodeid tõhusamaks ja optimaalsemaks muuta. Käesolev doktoritöö keskendub ülalnimetatud funktsionaalse analüüsi meetodite tõhustamisele. Selleks on välja pakutud erinevaid meetodeid ning nende efektiivsus on eksperimentaalsete uuringutega tõestatud ja avaldatud artiklites. Kõigepealt ühendati kolm rikete analüüsimisel kasutatavat tööriista, mille tulemusi võrreldi vastavalt standardile ISO 26262. Samas kasutati simulatsioonipõhise funktsionaalse testi asemel testigeneraatori poolt genereeritud testvektoreid ning rikete sisestamise tööriista, et kasutada ära testigeneraatori võimsust rikete tuvastamisel. Seega suurenes tuvastatud rikete hulk. Samuti arendati välja AutoSoC, mis kujutab endast openRISC-i protsessori arhitektuuril põhinevat avatud lähtekoodiga modulaarset kiipsüsteemi. AutoSoC sisaldab turvamehhanisme, mida soovitab ka ISO 26262 ning on terviklik platvorm, mille peal teadlased saavad enda väljatöötatud meetodeid proovida. AutoSoC on esitatud nii registersiirde kui ka loogikalülituste tasemel, sellega viidi läbi mõned funktsionaalohutuse analüüsid ja jagati tulemusi. Järgmisena optimeeriti simulatsioonipõhiseid rikete sisestamise kampaaniaid rikete loendi kärpimise kaudu: *(i)* Simuleeriti skeeme, analüüsiti katvuse tulemusi ning kriitilisi ajasamme ning seetõttu tuvastatud kriitilisi rikkeid, siis viidi läbi rikete sisestamise kampaania identifitseeritud kriitiliste rikete jaoks. Tänu sellele vähenes rikete sisestamise kampaaniate läbiviimise aeg. *(ii)* Abstraktsioonitaseme muutmisega viidi rikete sisestus läbi ainult kriitilistele mäluelementidele, sisestades samaaegselt mitu riket. Sel viisil optimeeriti simulatsioonipõhiseid rikete sisestamise kampaaniaid, vähendades rikete hulka. Lisaks viidi AutoSoC skeemil läbi ohutute rikete tuvastamine, kui see töötab CCA rakendusega oma tööea jooksul. Selles doktoritöös, mis simuleerib realistlikku rakendust, määrati riistvarakoodi katvuse analüüsi abil need osad, mida CCA rakenduses ei kasutata. Teisisõnu analüüsiti CCA käitumist ja seejärel teisendati see käitumine automaatselt formaalseteks omadusteks. Lõpuks, AutoSoCis viidi läbi turvaliste rikete tuvastamine, kui see töötab oma tööea jooksul CCA-ga. Lühidalt kokku võttes, käesolev doktoritöö optimeerib ja muudab tõhusamaks autotööstuse kiipsüsteemide analüüsimeetodeid.

Appendix 1

I

A. C. Bagbaba, F. Augusto da Silva, and C. Sauer. Improving the confidence level in functional safety simulation tools for iso 26262. In *2018 Design and Verification Conference (DVCON)*, 2018

Improving the Confidence Level in Functional Safety Simulation Tools for ISO 26262

Ahmet Cagri Bagbaba, Cadence Design Systems GmbH, Feldkirchen, Germany (*abagbaba@cadence.com*)

Felipe Augusto Da Silva, Cadence Design Systems GmbH, Feldkirchen, Germany (*dasilva@cadence.com*)

Christian Sauer, Cadence Design Systems GmbH, Feldkirchen, Germany (*sauerc@cadence.com*)

***Abstract*— Higher Tool Confidence Level (TCL) is needed for tools used on the verification of safety-critical SoCs, aiming to achieve the required Automotive Safety Integrity Level in ISO 26262. This paper presents a methodology to improve the confidence level of functional safety verification flow. To do this, we compare the fault-list generated by the fault injection (FI) simulator with the Automatic Test Pattern Generation (ATPG) flow for stuck-at (SA) fault types. Moreover, we compare fault coverage results by using test vectors generated by the ATPG tool so the result of the FI simulation is compared to the results gained from the ATPG. This is a way to improve simulator's confidence level by taking advantage of strength of the ATPG.**

Keywords—iso26262; functional safety; tool confidence; atpg; dft

I. INTRODUCTION

Functional safety (FS) refers to the absence of unreasonable risk caused by systematic failures and random hardware failures [1]. FS and especially the analysis of random failures are becoming part of the requirements for the design of complex systems. Therefore, a tighter integration between FS analysis and the standard platform design and verification is required. To achieve the FS of SoCs it is important analyze the use cases for all the flow tools according to their probability of introducing errors. This analysis shall evaluate if the malfunctioning tool or its erroneous output can lead to the violation of a safety requirement [2]. Based on this analysis, ISO 26262 Part 8 covers all aspects of TCL and defines key concepts of confidence and qualification [1]. The TCL assesses the error injection risk of each tool in the flow to document the confidence level for the data processing of each tool.

In this work, we increase the TCL of FI simulator tool. One way for accomplishing this is to compare results of FI simulator with the well-known and trusted results from ATPG. The purpose is to make FI simulator as optimal as the ATPG tool. We focus on two main aspects: The first is to assure that the instrumentation of the ATPG tool and the FI simulator is equivalent i.e. all optimizations are used. The second is to compare fault coverage results reported by the ATPG tool and the FI simulator. Developed methodologies are demonstrated as proof-of-concepts within the Cadence Design & Functional Safety flow [3] and the context of the ISO 26262.

The rest of this paper is organized as follows: in Chapter II, we mention about related works. Chapter III includes overview information about functional safety, fault injection simulation and the ATPG. We explain our approach and show the results in Chapter IV and V. In the final chapter, we conclude this paper by summarizing the work.

II. RELATED WORK

According to the ISO 26262 [1], verification tools employed on safety-critical automotive embedded systems must undergo qualification. In [4] authors present a semi-automatic qualification method involving a monitor and fault injection that reduce cost in the qualification process. In [5], authors define equivalence and dominance relations between fault pairs to optimize fault lists. In [6], authors tie functional safety to the traditional EDA domain.

Our approach, beside literature, uses results of the ATPG to prove the confidence of FI simulator and evaluate the tool potential impact on safety applications and the tool error detection capabilities. This means that same fault list is generated and most of the results are same with the ATPG tool.

III. Overview

A. *Functional Safety and ISO 26262*

Functional safety is the automotive industry standard, designed for safety-related systems for series production passenger vehicles with a maximum gross vehicle mass up to 3500 kg and that are equipped with one or more electrical/electronic subsystems [7]. Malfunction of the electrical/electronic component is classified into two types as systematic failures and random failures. Systematic failures represent the failures in an item or function that are induced a deterministic way during development, manufacturing or maintenance. Random failures represent failures during the lifetime of a hardware element. They can be classified as permanent faults (stuck-at faults) and transient faults (single-event-upsets or soft errors). Our focus in this work is permanent faults.

The design of safety systems involves error correction using checkers and error correction using redundancy. The former defines checkers which monitor the systems and trigger error response and recovery features when necessary. The latter defines redundancy which involves duplication of the entire system or a portion of the system. All requirements must be implemented by tracing from the system to components and ensured their development flow aligns with a tool confidence level. Also, recording and reporting functional safety measures to have a verified system are important.

B. *Fault Injection Simulation*

Fault injection enables to verify the capability of a safety mechanism to recognize failures in a design’s functionality, by injecting faults into the design. One way of doing this is fault injection simulation. In fault injection simulation, target system and the possible hardware faults are modeled and simulated by the simulator. In this process, the system behaves as if there is a hardware fault. The advantage of the fault injection simulation is that there is no risk to damage the system in use. Moreover, it is cheap in terms of time and effort. Additionally, fault injection simulation has a high observability and controllability in presence of faults.

To inject faults into a design, the FI simulator needs to know the fault target at which to inject fault. In this work, we enable fault instrumentation on the ports of cell instances. Therefore, all library cell ports are identified as faultable within the specified instance or module. This is equivalent to using the ATPG semantics.

There are different types of fault model to inject on specified fault nodes. The FI simulator supports single event upsets, stuck-at-0/stuck-at-1 and single event transient. We use stuck-at fault models in this work. The stuck-at model forces a signal to either 0 or 1 from the start of fault injection through to the end of simulation. This model can apply to nets or registers. The fault injection simulation flow starts with invoking of elaborator and instrumenting of faults according to a fault target. Then, a good simulation is run to generate reference values for fault simulation. Additionally, strobe points are specified in this point to monitor signals. Finally, one or more fault simulations with faults injected are run and generated reports are analyzed by the help of functional outputs (F-O) and checker outputs (C-O). Single run is illustrated in Figure 1.

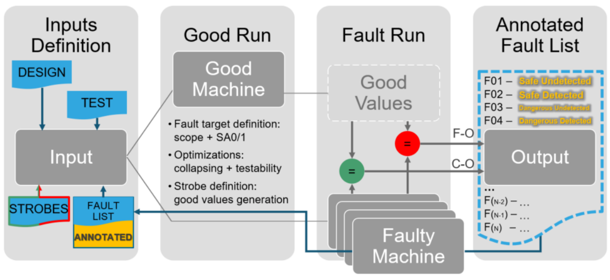


Figure 1-Fault Injection Simulation

C. *Automatic Test Pattern Generation*

Testability has become a critical concern for ASIC designers. Design-For-Test (DFT) techniques provide measures to test the manufactured device for coverage and quality. We use here Scan-based test which is one of the DFT techniques. This method basically replaces D flip flops with their scan-equivalent flops and serially connects the scan flops into scan chains. By replacing the flip-flops with their scan-equivalent flip-flops, the ATPG tool can achieve higher fault coverage and generate a more compact test pattern set for the design.

Defining a testmode for ATPG is an important step. A testmode is a specific test configuration of the design. The configuration defines how the test structures are accessed and how clocking is controlled. Each configuration is defined by the clocking and other test function pins, and the current test methodology. In this work, we use FULLSCAN testmode. The purpose of this testmode is to get static test using no compression.

Another step of the ATPG is generation of test vectors. ATPG tool can write test vectors to meet the manufacturing interface needs of IC manufacturers in different formats such as Standard Test Interface Language, Waveform Generation Language, and Verilog. In this work, we use Verilog test vectors.

IV. Approach

ISO 26262 tool confidence level can be improved by use of redundant methodologies to detect errors in the tool outputs. Cadence offers different technologies capable of generating fault list and ATPG from the Cadence® Modus™ DFT Software Solution can work together with the Cadence® Xcelium™ Fault Simulator (XFS) to create robust and optimized fault injection campaign. In this work, XFS for FI simulation, Cadence® Genus™ Synthesis Solution for synthesis and Modus for ATPG are used. Our purpose is to compare two different approaches regarding the results and improve the FI simulator's confidence. Comparison of the results to a different approach increases TCL and provide more optimized fault list. In other saying, we show that different tools capable of generating fault list bring about same fault list. Scan-based Design-for-Test structure is inserted during the synthesis. We have two methodologies explained below.

A. *Instrumentation comparison between the ATPG tool and FI simulator*

Fault-list can be generated both by the ATPG tool and FI simulator. ATPG tool is accepted as reference because of its usage for long years in industry. Hence, it is expected that FI simulator's fault optimization achieves the same capabilities as the ATPG tool. The purpose is to prove that FI simulator contains all instrumentation and optimization potential. In other saying, the aim is to show that instrumentation of FI simulator and ATPG tool is equivalent.

This approach is shown in Figure 2 as Compare Optimized Fault Lists. Here, the fault lists generated by both tools compared to verify: whether the same faults are instrumented, if they have the same number of prime faults (collapsing), and if they have the same number of untestable faults. Our methodology finds fault collapsed groups and differences between lists if there exists. For this comparison, the designs are synthesized with DFT insertion. Elaboration of gate level (GL) Design Under Test (DUT) is enough in the simulator side. For the ATPG side, test patterns are generated by the ATPG tool after scan-chain insertion. The ATPG tool creates logic/scan test and produce results. All comparisons are done for stuck-at faults.

B. *Compare fault coverage results reported by the ATPG tool and FI simulator*

ATPG is a process used in SoCs testing wherein the test vectors required to check a device for faults. Test vectors are automatically generated by the tool. However, testbench is required for FI simulator. Therefore, it is a good idea to use test vectors generated by the ATPG tool as a test instead of writing a testbench.

The FI simulator can be used in conjunction with ATPG. In this way, each fault detected by the ATPG tool should be detected by FI simulator. The aim is to prove that FI simulator and the ATPG tool have the same coverage.

In this step, DFT is inserted by the synthesize tool and the ATPG tool is used to collect all SA faults at GL netlist and generate test patterns as shown in Figure 2 as Compare Annotated Fault Lists Level. All SA faults instrumented by the ATPG tool are injected by FI simulator. By the help of this, ATPG tool annotated fault list and FI simulator annotated fault list can be easily compared and fault coverage differences can be observed.

ATPG results include different annotation types such as tested, untested, ignored and collapsed. Beside this fault injection simulation results have annotation results such as detected, untestable, and undetected. To make a comparison between results, we define a comparison method shown in Table-I. Second row of the table explains that if the fault is tested in Modus, it needs to be detected in Xcelium. In this case, check annotation result is PASS. Last column shows the annotation result whether it is pass or warning. WARNING means that there is a difference between results. This method helps for debugging in case of difference between results and tool verification for ISO 26262.

Table I-Annotation Results Comparison Method

Xcelium Annotation Types	**Modus Annotation Types**	**Check Annotation**
Detected	Tested by simulation	PASS
Untestable	Ignored (unclassified)	PASS
Potentially Detected	Tested by implication	PASS
Undetected	Collapsed Tested by simulation	WARNING
Untestable	Collapsed Tested by simulation	WARNING

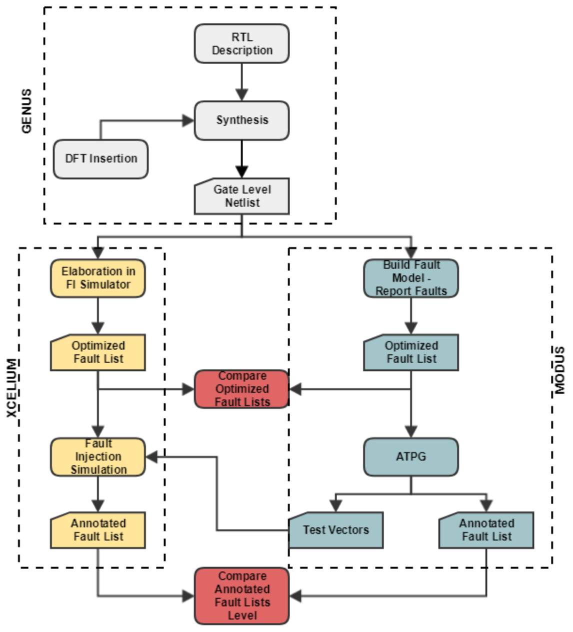


Figure 2-Flow

V. Results

First, we show that ATPG and fault injection simulator generate the same fault lists. This provides us to compare results easily. All circuits are analyzed for SA0 and SA1 fault models. Collapsed and untestable faults are included for analyses.

Table-II shows the results of instrumentation comparison between Modus and XFS. This comparison is shown in Figure 2 as compare optimized fault lists. IWLS 2005 benchmark circuits [7] are used to collect results. Optimized fault list includes faults that do not need to be simulated since behavior of the design in presence of these faults can be predicted. First row of Table-II gives the number of instrumented faults for each design. Last row of this table shows the number of optimization differences. The reason of this is different collapsing approach of tools. In other saying, collapsing methods of tools have effect on the results. For example, Modus does not collapse primary inputs; however, primary inputs are collapsed by XFS. This does not change functionality of circuits.

Table II-Comparison of optimized fault lists

	ac97	**aes**	**dma**	**total**
Nr of instrumented faults	57226	91916	66708	212850
Nr of optimization differences	12	4	76	92

Table-III shows the results of the second part of this work which is comparison of fault coverage results and annotated fault list level reported by tools. Here, we use ITC'99 benchmark circuits [8] to generate results. For each fault listed in the ATPG annotated fault list, faulty behavior is simulated, and the observation points are compared against the reference values generated during the good run as shown in Figure 1. This flow is shown in Figure 2 as compare annotated fault lists level. For each tested design, number of total faults in Xcelium and Modus can be seen in Table-III. Xcelium columns shows the number of faults after fault simulation with ATPG test vectors. Last column shows the comparison results that is explained in Section IV. Overall results point that although test vectors generated by Modus is for external tester devices, we obtain same fault coverage values when we applied test vectors into the fault injection simulation tool. The length of test vectors is important here. It must be set to maximum length in order to obtain correct results.

Table III- Comparison of annotated fault list level

Design	**Xcelium**	**Modus**	**Check Results**
b01	230	230	PASS
b02	174	174	PASS
b03	882	882	PASS
b04	2268	2268	PASS
b05	2512	2512	PASS
b06	274	274	PASS
b07	1882	1882	PASS
b08	766	766	PASS

VI. CONCLUSIONS

FS becomes a crucial requirement in the safety critical automotive systems. This is causing a shift in the traditional design flow and pushing ISO26262 compliance into the traditional EDA tools. Therefore, it is vital to improve TCL. For this purpose, we propose methodologies to improve TCL in FI simulator by using strength of the ATPG. By considering guidance of ATPG results, we show that FI simulator contains all instrumentation/optimization potential and they have same coverage results. In the first method, optimized fault lists are compared. Results show that there are differences between the ATPG and FI simulator due to collapsing approach of tools, but this does not affect functionality of the circuits. In the second part of this paper, we compare annotated fault list level of the tools by using test vectors generated by the ATPG tool as a test in FI simulation. This method shows that both tools have same annotated fault list level and coverage values.

ACKNOWLEDGEMENT

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

REFERENCES

[1] ISO, “ISO26262-road vehicles functional safety,” International Organization for Standarization in ISO26262, 2011.

[2] M. Conrad, P. Munier, and F. Rauch, "Qualifying Software Tools According to ISO 26262," (white paper) http://www.mathworks.se/automotive/ standards/iso-26262.html, 2010, The Mathworks, Inc.

[3] Meeting Functional Safety Requirements Efficiently Via Electronic Design Tools and Techniques, [online] Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/solutions/automotive-functional-safety-wp.pdf

[4] Q. Wang, A. Wallin, V. Izosimov, U. Ingelsson and Z. Peng, "Test tool qualification through fault injection," 2012 17th IEEE European Test Symposium (ETS), Annecy, 2012, pp. 1-1.

[5] I. Pomeranz and S. M. Reddy, "Equivalence, Dominance, and Similarity Relations between Fault Pairs and a Fault Pair Collapsing Process for Fault Diagnosis," in IEEE Transactions on Computers, vol. 59, no. 2, pp. 150-158, Feb. 2010.

[6] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, 2017, pp. 970-975.

Beckers, Kristian, et. Al. “Systematic derivation of functional safety requirements for automative systems.” *International Conference on Computer Safety, Reliability, and Security*. Springer, Cham, 2014.

[7] Cadence Research Berkeley, “International Workshop on Logic and Synthesis (IWLS) 2005 Benchmarks”.

[8] F. Corno, M. S. Reorda and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," in IEEE Design & Test of Computers, vol. 17, no. 3, pp. 44-53, July-Sept. 2000.

Appendix 2

II

F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer. Use of formal methods for verification and optimization of fault lists in the scope of iso26262. In *2018 Design and Verification Conference (DVCON)*, 2018

Use of Formal Methods for verification and optimization of Fault Lists in the scope of ISO26262

Felipe Augusto da Silva, Cadence Design Systems GmbH, Feldkirchen, Germany (*dasilva@cadence.com*)

Ahmet Cagri Bagbaba, Cadence Design Systems GmbH, Feldkirchen, Germany (*abagbaba@cadence.com*)

Said Hamdioui, Delft University of Technology, Delft, The Netherlands (*s.hamdiouia@tudelft.nl*)

Christian Sauer, Cadence Design Systems GmbH, Feldkirchen, Germany (*sauerc@cadence.com*)

***Abstract*—This work aims at an alternative method to verify the correctness of Fault Lists generated by fault simulators tools in context of safety verification. The lists generated by simulation tools are verified against lists from formal tools. The consistency evaluation between the lists supports the Tool Confidence Level (TCL) assessment, defined in the ISO26262. In addition, formal tools have the potential of performing optimization in Fault Lists by annotation of the expected behavior of the design under fault. Our work demonstrates the feasibility of using Formal Methods to verify and optimize the fault list from simulators. Results indicate an average reduction of 29.5% on the number of faults to be simulated and demonstrate that it is possible to achieve TCL by verification of the fault lists.**

Keywords—ISO26262; Fault Injection; Formal; Simulation; Tool Qualification.

I. Introduction

With the increasing complexity in automotive applications such as autonomous driving, the use of electronics systems in this domain is growing exponentially. This is causing a shift in the traditional design flow and is pushing ISO26262 compliance down to the semiconductor chain. As a result, Functional Safety compliance becomes part of the requirements for the development of complex systems. During the development of an Integrated Circuit (IC) compliant with ISO26262, one of the critical tasks is the evaluation of the effectiveness of the design to cope with random hardware failures. This is usually done by execution of Fault Injection (FI) Simulations, where each possible fault candidate of the design is evaluated for robustness to random faults, and the behavior of the design under these faults is simulated. In complex designs, where millions of design components are susceptible for random faults, this process becomes challenging [1].

To facilitate FI Simulation campaigns, EDA tools may be used for automation of behavioral analysis of a design. By examining the description of a design, simulation tools are able to identify what design components should be considered for fault injection and simulate the behavior of the design under the effect of these faults. The provided automation increases the possibility of faults being introduced or masked by malfunction in the tool. Aiming to avoid these malfunctions, ISO26262 includes instructions for qualification of tools. Any tool that supports activities required by the standard, must be evaluated to show the minimum level of confidence necessary for the intend activities.

The level of confidence of a tool is determined by evaluating the possibility of a malfunctioning to introduce or fail to detect errors in the design under development. In the case of a tool used in FI Simulation, a malfunction could mask or introduce an error on the fault candidates and on the analysis of the behavior of the faulty design. To guarantee the confidence in the results generated by the tool, an evaluation methodology is required. The outputs of the tool should be verified to prevent or detect any malfunctions.

Considering the automation provided by EDA tools on FI Simulations, this work focuses on improving the Level of Confidence on the Fault Lists generated by simulators with Formal Methods. In addition, the results from formal analysis allowed us to optimize the Fault Lists and reduce the time of FI Simulation campaigns.

II. Related Work

The challenges of tool qualification per ISO26262 are exemplified in [2]. The authors present a semi-automatic qualification method for a verification tool that can reduce costs in the qualification process. The work highlights the importance of applying automatic verification on the outputs of a tool, to decrease the efforts of manual verification. In [3], formal methods are used to determine the behavior of a design under fault. The authors propose a fault injection model that allows the verification of a design by symbolic simulation. A mixed approach using formal methods with simulation to decrease the time of fault injections campaigns, is explored in [4]. Formal is used to show that a failure state is not achievable with the injected fault, thus the simulation can be stopped. Different works are employing combined fault injection analysis flows with simulation and formal methods, [5] [6][7]. The strength of formal methods, in analyzing the behavior of a design to all test stimulus, is applied to leverage the most appropriate setups for the simulation campaigns.

Our approach combines simulation and formal methods as a methodology for verification of the results generated by both tools. Formal analysis is used to verify Fault Lists generated by the simulator, thus increasing the confidence in the tool outputs, as required by ISO26262. To the best of our knowledge this approach was not previously used. In addition, as seen in other works, formal analysis can reduce the number of required simulations by pre-evaluating the fault propagation potentials.

III. Fault Injection Campaigns

ISO26262 requires that any component that implements a safety related functionality, reach a minimum level of tolerance to random hardware failures. Coverage for this type of failure is usually increased by addition of Safety Mechanisms to the design. Safety Mechanisms should be able to guarantee that fault propagation cannot disturb a safety related functionality.

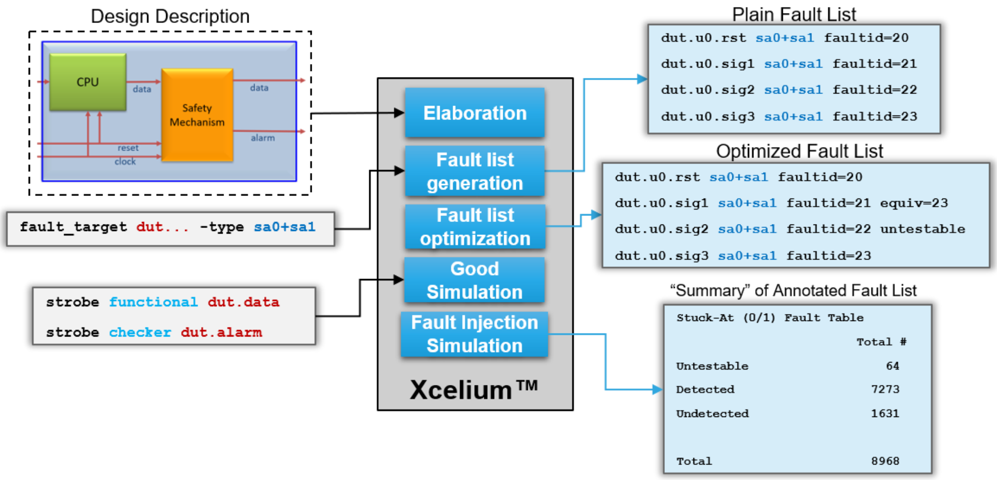
The effectiveness of the design to cope with random hardware failures should be quantitatively demonstrated as defined by the standard. To accomplish this, it is necessary to assess the efficiency of the Safety Mechanisms to handle critical faults thus allowing to achieve targeted safety metrics. Fault Injection Simulation is a widely used technique to perform this analysis being the method recommended by ISO26262.

A. *Fault Injection Simulation*

Analysis of Fault Injection by Simulation is widely used and available in a variety of tools. These tools are able to analyze a Register Transfer Level (RTL) or Gate-Level (GTL) descriptions of a design and, based on given test inputs, simulate their behavior. The effect that a fault causes in the design is determined by comparing the behavior of the design with and without faults. The selection of the tool must consider the available features and aspects of performance, as FI Simulation campaigns can become long as design complexity increases. Our work deploys Cadence® Xcelium™ Fault Simulator (XFS) to perform the Fault Injection Simulation [8]. The flow implemented by XFS for Fault Injection Simulation is as follow:

1. Elaboration of RTL/GTL design description.
2. Fault List Generation: fault node candidates found in the design are listed for each available fault type. User should define rules (e.g. all signals) to identify fault node candidates and fault types (e.g. Stuck-at-0 (SA0) and Stuck-at-1 (SA1)). Information is stored in a Plain Fault List.
3. Fault List Optimization: Plain Fault List is analyzed to identify faults that do not need to be simulated as the behavior of the design in presence of these faults can be predicted. Information is stored in an Optimized Fault List.
4. Good Simulation: fault-free behavior of design is simulated. The user should define observation points in the design to identify: (1) Fault propagation to a functional output: functional strobes; (2) Activation of the Safety Mechanism: checker strobes. Strobe values during good simulation are stored.
5. Fault Injection Simulation: For each fault, listed in the Optimized Fault List, the design faulty behavior is simulated, and the observation points compared against the reference values generated during Good Simulation. Results of FI are stored in the Annotated Fault List.

Looking on the perspective of Tool Qualification, there are three main outputs of the Simulation tool that should be verified: The Plain Fault List, the Optimized Fault List and the Annotated Fault List. Figure 1 illustrates the FI Simulation flow with the required user inputs and described outputs.



```
fault_target dut... -type sa0+sa1
```

```
strobe functional dut.data
strobe checker dut.alarm
```

```
dut.u0.rst sa0+sa1 faultid=20
dut.u0.sig1 sa0+sa1 faultid=21
dut.u0.sig2 sa0+sa1 faultid=22
dut.u0.sig3 sa0+sa1 faultid=23
```

```
dut.u0.rst sa0+sa1 faultid=20
dut.u0.sig1 sa0+sa1 faultid=21 equiv=23
dut.u0.sig2 sa0+sa1 faultid=22 untestable
dut.u0.sig3 sa0+sa1 faultid=23
```

```
Stuck-At (0/1) Fault Table
                        Total #
Untestable                   64
Detected                   7273
Undetected                 1631

Total                      8968
```

Figure 1. Xcelium Fault Injection Simulation Flow.

Although, FI simulation is the recommended method for FI analysis, the process of simulating each single Fault is highly costly. As the behavior of the design is simulated with single stimulus at a time, there is a considerable chance that faults will not propagate to a strobe, in other words will be Undetected for this specific stimulus. Defining the required group of stimuli to assure that every single fault will propagate to a strobe is nearly impossible in complex designs. To address these challenges, different technologies are being analyzed to decrease the efforts of FI analysis. The use of Formal Methods is a promising solution, being already deployed by different vendors in their Formal Solutions.

B. Fault Analysis by Formal Methods

While FI simulation is limited to a single context, applying a single stimulus using a single fault model, formal fault analysis is not limited to a specific time or state. Instead, the context is global, and every evaluation context, stimulus and faults, are considered. Consequently, formal analysis can exhaustively prove that an Untestable fault can never propagate to a strobe. If there is no possibility of propagation, the fault can be considered Safe and do not need to be simulated.

Different vendors are implementing FI Analysis capabilities in their Formal Solutions, this work uses the Functional Safety Verification (FSV) application from Cadence JasperGold® (JG) Formal Verification Platform. JG FSV requires no formal languages knowledge, as all required properties are automatically generated by the tool. Fault Analysis is available in a standalone mode, but also includes build-in support for integration with XFS, allowing the deployment of both tools in a unified FI Analysis flow. JG FSV includes two main fault analysis techniques, Standard Analysis and Advanced Analysis [9].

The Standard Analysis verify the testability of faults. It is an automated pre-qualification flow for simulation that improve the results of the Optimized Fault List. FSV applies structural fault analysis techniques to verify if the injected faults could affect the results on one of the strobes. In addition, the fault list is optimized by Fault Relations Analysis. The tool analyzes the design for relationship between fault pairs in which the result of one fault can be predicted by the behavior of the other. Fault pairs are then included in the same Collapsing Group. The behavior of all Collapsing Group is predicted by simulation of only one representative of the group, called the Prime Fault.

The Advanced Analysis deploys formal propagability and activation analysis. Activation Analysis checks whether the fault can be functionally activated from the inputs. Propagation Analysis checks whether the fault can propagate to a strobe. If it cannot, then it is determined to be Safe. If it can, this analysis will identify the necessary stimulus for propagation. The strobes can be functional or checker. Propagation of the fault to a functional strobe can lead to a functional safety violation, while propagation to a checker strobe indicates that the Safe Mechanism detected the fault. Figure 2 illustrates JG FSV Fault Injection Advanced Analysis flow. Properties to verify the propagation effects from faults and strobes detection are automatically generated, and then verified to all possible input stimulus. Results are compared against a copy (Bad Machine) of the design were the fault is injected.

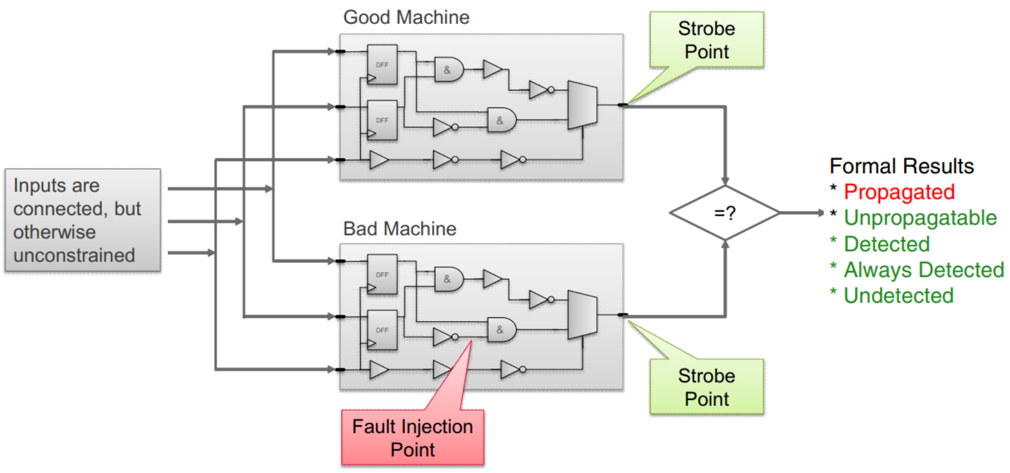


Figure 2. Jasper Gold FSV Fault Injection Analysis Flow.

The different strengths of Simulation and Formal can complement each other for FI Analysis. The combined flow allows reduction of simulation efforts by increasing fault optimization, identifying propagation potentials and by identifying stimulus that will cause fault propagation during simulation.

The build-in integrated flow allows deployment of JG FSV Standard Analysis on the Optimized Fault List from XFS. The formal analysis will reduce the number of faults to be simulated by leveraging formal results for Safe Faults and Collapsing Groups. After simulation, JG FSV Advanced Analysis can be executed on the remaining Undetected faults to verify if they can propagate to a strobe and what is the required stimulus.

The FI Analysis from JG FSV can generate the same outputs that are generated by XFS. JG FSV flow starts by Analyzing and Elaborating a design description. Next, user should set the fault type and design candidates for fault injection, generating a Plain Fault List. After, the Standard Analysis will generate an Optimized Fault List, including Safe and Collapsing information. And last, the Advanced Analysis will generate an Annotated Fault List by including information about propagation and detection of faults. By using formal to generate the same outputs from the simulator, it is possible to automatically verify the consistence between the results. As stated by the ISO26262, prevention or detection of tool malfunctioning can be accomplished through redundancy in software tools [10].

IV. Verification and Optimization Flow

Even though the build-in integrated flow between Xcelium and JG facilitates the FI Analysis, from the perspective of tool qualification, it is preferable to run both tools in standalone mode. To use the outputs from formal to verify the outputs from the simulator, it is necessary to show that there is no interference between the tools. As during the integrated flow, the tools share the same fault database, we have decided to separate the flows.

To automate the evaluation of the outputs generated by both tools, a Build Manager application was developed. For each given design, the application automates the elaboration and analysis of the design, on both tools, and controls the execution flows, including the formal analysis in JG FSV and FI simulations on XFS. Finally, the relevant data is retrieved from both tools and compared. The comparison between the lists is based on rules that associate the annotations used by the tools. For example, faults classified as Untestable by XFS are equivalent to faults classified as Safe by JG FSV.

A detailed report is generated to allow review of the results. An error in an output caused by a malfunction in one of the tools, can be detected by the annotation association rules and could be verified in the detailed report. For example, if XFS simulation annotates a fault as Detected and JG FSV annotates the same fault as Safe, this would indicate a malfunction in one of the tools. A sample of the detailed report, including an example of a tool malfunction, is illustrated in Table I.

Table I. Detailed Report Example

	XFS				JG FSV				Result
Fault ID	Signal Name	Fault Type	Annotation	Collapsing	Signal Name	Fault Type	Annotation	Collapsing	
0	dut.u0.rst	sa0	Dangerous		dut.u0.rst	SA0	Propagated		PASS
1	dut.u0.rst	sa1	Untestable		dut.u0.rst	SA1	Safe		PASS
2	dut.u0.sig1	sa0	Detected		dut.u0.sig1	SA0	Detected		PASS
3	**dut.u0.sig1**	**sa1**	**Detected**		**dut.u0.sig1**	**SA1**	**Safe**		**WARNING**
4	dut.u0.sig2	sa0	Dangerous	equiv=2	dut.u0.sig2	SA0	Propagated	2	PASS
5	dut.u0.sig2	sa1	Detected		dut.u0.sig2	SA1	Detected		PASS

V. Results

Results were collected by executing the Build Manager application on a set of selected designs from the IWLS 2005 benchmarks [11]. The benchmark contains a collection of RTL and Gate Level description of 84 different designs, varying from small cores to complete System-on-Chip.

Initially, the RTL description of 16 designs were analyzed for Stuck-at-0 and Stuck-at-1 faults. Fault Lists generated by both tools are analyzed to verify: (1) The tools generated the same faults, (2) Which faults are annotated as “Safe”, (3) Which faults are collapsed and (4) All annotations respected the association rules.

Fault Injection analysis with simulation requires that different test stimuli is applied to assure propagation of each fault to a determined strobe. If a fault does not propagate, it is considered Undetected. For the scope of this work, as the idea is not to achieve full verification of the example designs, test vectors were not further developed and complete FI Simulation was not executed. Therefore, some faults are annotated as “Not Injected” by XFS and as “Unknown” by JG FSV. These faults are not considered as a tool malfunction, as they should be revaluated after full verification environment is set-up.

The results of the benchmark evaluation are shown in Table II. For each tested design, the total number of faults and Safe annotations for each tool are illustrated. The column Fault List Reduction highlights the fault reduction percentage per design, when including the JG annotation to the XFS Fault List. JG FSV Standard formal analysis run time in seconds is demonstrated in the corresponding column.

Table II. Summary of Results

Design	XFS		Jasper Gold				Fault List Reduction	
	Nº of Faults	*Safe Faults*	*Nº of Faults*	*Safe Faults*	*Collapsed Faults*	*Run time (s)*	*by Safe Faults*	*by Collapsed Faults*
DMA	33428	106	33428	4921	8734	186	14,40 %	26,13 %
ac97	11192	134	11192	1401	2326	674	9,88 %	20,78 %
aes	4266	0	4266	49	1408	168	1,15 %	33,01 %
i2c	528	0	528	14	86	9	2,65 %	16,29 %
mem_ctrl	11044	8	11044	3933	2246	346	34,75 %	22,11 %

Design	XFS		Jasper Gold				Fault List Reduction	
	Nº of Faults	*Safe Faults*	*Nº of Faults*	*Safe Faults*	*Collapsed Faults*	*Run time (s)*	*by Safe Faults*	*by Collapsed Faults*
sasc	86	0	86	1	0	7	1,16 %	0,00 %
simple_spi	534	28	534	35	54	9	1,31 %	10,11 %
spi	1396	0	1396	12	324	13	0,86 %	23,21 %
ss_pcm	242	2	242	3	1	7	0,41 %	0,41 %
systemcaes	9302	0	9302	425	2664	40	3,37 %	47,38 %
systemdes	4104	64	4104	98	1806	41	0,77 %	47,84 %
tv80	1942	36	1942	51	206	49	0,73 %	15,48 %
usb_funct	20386	56	20386	8128	6483	665	39,38 %	32,17 %
usb_phy	364	0	364	3	58	8	0,80 %	18,62 %
vga_lcd	762	0	762	4	0	9	0,52 %	0,00 %
wb_conmax	106666	0	106666	2794	65216	186	2,61 %	61,31 %

VI. Conclusions

The combination between simulation and formal methods is becoming a stablished method for Fault Injection Analysis and appears as a promising practice for verification of Fault Lists. Looking at XFS and JG FSV as representatives of these technologies, we propose an alternative methodology for the evaluation of Fault Lists on the scope of ISO26262. Inclusion of redundancy as a method to detect malfunctions in a tool, is one of the standard suggested methods for achieving Tool Confidence Level. In addition, the formal methods applied by JG, provide improved information about the propagation effects of the faults, allowing the optimization of the Fault List, and therefore, reducing the number of required faults to be simulated. Preliminary results have shown that the Fault List of both tools are equivalent, allowing the use of JG to verify the outputs of XFS. Furthermore, the results from JG allowed an average reduction of 29.5% on the number of faults to be simulated.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 722325.

References

[1] Y.C. Chang, L.R. Huang, H.C. Liu, C.J. Yang and C.T. Chiu, "Assessing automotive functional safety microprocessor with ISO26262 hardware requirements", 2014 International Symposium on VLSI Design, Automation and Test (VLSI-DAT).

[2] Q. Wang, A. Wallin, V. Izosimov, U. Ingelsson and Z. Peng, "Test tool qualification through fault injection", 2012 17th IEEE European Test Symposium (ETS).

[3] U. Krautz, M. Pflanz, C. Jacobi, H.W. Tast, K. Weber and H.T. Vierhaus, "Evaluating Coverage of Error Detection Logic for Soft Errors using Formal Methods", Proceedings of the Design Automation & Test in Europe Conference, 2006, vol 1.

[4] A. Bernardini, W. Ecker and U. Schlichtmann, "Where formal verification can help in functional safety analysis", 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).

[5] K. Devarajegowda and J. Vliegen, "Deploying Formal and Simulation in Mutual-Exclusive Manner using JasperGold's Proofcore Technology", Cadence User Conference CDNLive EMEA 2017.

[6] S. Marchese and J. Grosse, "Formal Fault Propagation Analysis that Scales to Modern Automotive SoCs", 2017 Design and Verification Conference and Exhibition (DVCON) Europe.

[7] A. Traskov, T. Ehrenberg and S. Loitz, "Fault Proof: Using Formal Techniques for Safety Verification and Fault Analysis", 2016 Design and Verification Conference and Exhibition (DVCON) Europe

[8] Cadence Design Systems, "Xcelium Fault Simulator User Guide", Product Version 2018.03

[9] Cadence Design Systems, "JasperGold Functional Safety Verification App User Guide", Product Version 2018.03

[10] International Organization for Standardization, "ISO26262 - Road Veichles - Functional Safety - Part 8: Supporting processes".

[11] Cadence Research Berkeley, "International Workshop on Logic and Synthesis (IWLS) 2005 Benchmarks".

Appendix 3

III

Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Efficient methodology for iso26262 functional safety verification. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 255–256, 2019

Efficient Methodology for ISO26262 Functional Safety Verification

Felipe Augusto da Silva[1,2], Ahmet Cagri Bagbaba[1], Said Hamdioui[2] and Christian Sauer[1]
[1]Cadence Design Systems, Feldkirchen, Germany
[2]Delft University of Technology, Delft, The Netherlands

***Abstract*— Tolerance to random hardware failures, required by ISO26262, entails accurate design behavior analysis, complex Verification Environments and expensive Fault Injection campaigns. This paper proposes a methodology combining the strengths of Automatic Test Pattern Generators (ATPG), Formal Methods and Fault Injection Simulation to decrease the efforts of Functional Safety Verification. Our methodology results in a fast-deployed Fault Injection environment achieving Fault detection rates higher than 99% on the tested designs. In addition, ISO26262 Tool Confidence level is improved by a fault analysis report that allows verification of malfunctions in the outputs of the tools.**

Keywords - ISO26262; Fault Injection Simulation; Formal Methods; ATPG; Functional Safety.

I. INTRODUCTION

Functional Safety Verification is one of the most challenging steps for Integrated Circuit (IC) compliance with ISO26262. In safety-critical applications the system must include Safety Mechanisms being able to detect up to 99% of the random faults susceptive of the design. In addition, ISO26262 requires that all possible malfunctions of tools (used during fault analysis) have to be considered as per Tool Qualification requirements [1]. Therefore, there is a high demand for effective Functional Safety Verification methodologies allowing the reduction of costs while maintaining the same levels of safety.

The commonly used method for Functional Safety Verification is Fault Injection (FI) Simulation [2][3]. The purpose is to show that fault effects can propagate to outputs and that Safety Mechanisms can detect them. In order to cause propagation of all faults, complex verification environments with numerous test inputs are required, resulting in long FI Campaigns. To address this challenge, we can deploy different verification technologies in a single methodology. Methodologies applying Formal Methods to identify faults that cannot propagate to outputs of the design (Safe faults) [4][5][6], and ATPG techniques to generate test patterns that potentialize fault propagation [7][8] have been proposed. Even though Simulation, Formal Methods and ATPG have complementary strengths, to the best of our knowledge, they were not previously combined in a single fault analysis flow that aims at fault propagation for compliance to ISO26262 requirements.

Our work takes advantage of three different technologies aiming to achieve high fault detection rates while decreasing the efforts of traditional FI Campaigns. ATPG is used for fast deployment of a verification environment that provides high fault propagation rate. The outputs from ATPG are used by the FI Simulator, for verifying the functional behavior of the design under each fault. In parallel, Formal Methods are applied to the design to identify Safe faults. In addition, the outputs of each tool are verified against each other to identify malfunctions, increasing the confidence in the tool's outputs, as required by ISO26262 [1]. The main contributions of our methodology are:

- Reduction of the efforts with Test Environment developments and in the number of required Simulations by deploying automatic generated ATPG Test Environments to FI Simulation campaigns.
- Increasing compliance to ISO26262 fault metrics by identification of Safe faults with formal methods.
- Generation of report containing detailed information of tool outputs to detect malfunctions.

II. PROPOSED METHODOLOGY

Our methodology aims to automate the execution of Simulation, ATPG and Formal analysis for a specific design. At the end of the execution, the outputs of the tools are compared to find discrepancies. An application was develop in order to control the execution flow and generate final reports. The Fault Checker application can be configured to use any ATPG, Simulation, and Formal tools. At the beginning of the execution, the user should configure the scripts to control the execution of each tool and provide the rules for parsing the tool reports.

The application starts with the execution of the ATPG and Formal flows. As these two flows are independent, they can be executed in parallel using different CPUs. Simulation flow requires the ATPG Testbench and test vectors to start. So, after the ATPG flow is finished, the Fault Checker will extract the generated Test Environment and will use it for the FI Simulation. At the end of each flow, the reports generated by the tools are parsed to a common format, allowing verification of the results to identify discrepancies between the tools. Finally, at the end of all flows, the relevant parsed data is retrieved and compared. The comparison is based on rules that associate the annotations used by each tool. For example, faults classified as Untestable by the Simulator are equivalent to faults classified as Safe by Formal and Ignored by ATPG. In case a rule is not obeyed, the Fault Checker will include a Warning tag to the report, informing that this fault requires attention from the designer.

This project has received funding from the European Unions Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 722325.

Results can be analyzed in a CSV report that details the annotation of each fault by each tool. An error caused by a malfunction in one of the tools, will be indicated by a Warning in the CSV report. For example, if simulation annotates a fault as Detected and Formal annotates the same fault as Safe, it indicate a malfunction in one of the tools. The report provides supplementary information with further possibilities for fault analysis. For example, if a specific fault is considered Undetected by Simulator and Dangerous by Formal, it means that formal analysis identified at least one test stimulus that can propagate the fault. This information can be used on a new FI Simulation to achieve detection of this fault. Any other discrepancy between the faults is indicated in the report with a Warning.

III. VALIDATION

This section describes the validation process of the proposed methodology. First the Fault Checker was configured with execution scripts to deploy Cadence®Xcelium™Fault Simulator (XFS), Cadence®JasperGold (JG) Formal Verification Platform and Cadence®Modus DFT Software Solution ATPG component as the representatives of each technology. Second, selected designs were verified by the Fault Checker. Table I details, for each design, the total number of faults, the fault detection rate, and the indication of Pass or Warning resulting from the verification of the tools by the Fault Checker.

TABLE I
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

During the Up Down Counter and Memories designs verification, the Fault Checker confirmed that all faults have equivalent annotations. As the examples are relatively simple, the different tools can determine that all faults can propagate to outputs. For the Memories design, the application detected 6 faults that were annotated as Safe by the Formal analysis, and can be disregarded.

On the AC97 design, the Fault Checker was able to detect 118 faults with distinctive annotations. From these, 49 faults were annotated as Safe by Formal and can be disregarded; 23 were annotated as Dangerous by Formal meaning that Formal can extract test inputs to cause propagation of the faults; and 46 faults were not classified, meaning that they require manual analysis.

During the analysis of the Conmax design, the methodology detected 263 discrepancies between the tools. From these, 7 faults were annotated as Dangerous by Formal. Meaning that results from Formal can be applied for detecting these faults during simulation. The other 256 faults have non conclusive annotations and should be manually analyzed.

The results demonstrated above corroborate with the listed contributions. First, the comparison of the fault classifications from each tool enables identification of tool malfunction. The report generated by the Fault Checker allows detailed analysis of faults and can be used to support ISO26262 Tool Qualification. Second, Safe faults classification by Formal Methods permits improvement of fault tolerance, by decreasing the total number of faults and improving ISO26262 metrics. Third, the proposed methodology shows considerable fault detection rates for all tested designs.

IV. CONCLUSIONS

Due to the harsh requirements for random hardware failures tolerance, Functional Safety verification is a challenging step for ISO26262 compliance. FI simulation, as part of this process, becomes a long and expensive procedure, that is usually repeated numerous times until the metrics for fault detection are achieved. We propose a methodology that deploys ATPG and Formal to support Simulation results and to decrease the overall effort of FI Simulations. Our methodology enables the use of test environments created with ATPG for the simulation of faults, and the use of Formal for identification of Safe faults. Formal results allow the optimization of the Fault List, reducing the number of faults to be simulated. In addition, the results of the tools are compared to identify discrepancies and potential defects. The inclusion of redundancy as a method to detect malfunctions in tools is suggested for achieving ISO26262 Tool Confidence [1]. Our results have shown high fault detection rates, achieving more than 99% of detected faults. In addition, the proposed methodology can identify Safe faults, contributing to reaching ISO26262 metrics.

REFERENCES

[1] ISO, *ISO 26262 - Road Veichles - Functional Safety - Part 8: Supporting processes*, International Standardization Organization Std., Nov. 2011.

[2] 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.

[3] 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.

[4] K. Devarajegowda and J. Vliegen, "Deploying formal and simulation in mutual-exclusive manner using jaspergolds proofcore technology," in *Cadence User Conference CDNLive EMEA*, 2017.

[5] S. Marchese and J. Grosse, "Formal fault propagation analysis that scales to modern automotive SoCs," in *2017 Design and Verification Conference and Exhibition DVCON Europe*, 2017.

[6] A. Traskov, T. Ehrenberg, and S. Loitz, "Fault proof: Using formal techniques for safety verification and fault analysis," in *2016 Design and Verification Conference and Exhibition DVCON Europe*. DVCON, 2016, pp. 27–32.

[7] 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.

[8] 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.

Appendix 4

IV

Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Efficient fault injection based on dynamic hdl slicing technique. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 52–53, 2019

Efficient Fault Injection based on Dynamic HDL Slicing Technique

Ahmet Cagri Bagbaba*†, Maksim Jenihhin†, Jaan Raik†, Christian Sauer*
*Cadence Design Systems, Munich, Germany; † Tallinn University of Technology, Tallinn, Estonia
Email: *{abagbaba, sauerc}@cadence.com, †{maksim.jenihhin, jaan.raik}@taltech.ee

***Abstract*—This work proposes a fault injection methodology where Hardware Description Language (HDL) code slicing is exploited to prune fault injection locations, thus enabling more efficient campaigns for safety mechanisms evaluation. In particular, the dynamic HDL slicing technique provides for a highly collapsed critical fault list and allows avoiding injections at redundant locations or time-steps. Experimental results show that the proposed methodology integrated into commercial tool flow doubles the simulation speed when comparing to the state-of-the-art industrial-grade EDA tool flows.**

***Index Terms*—Fault injection, fault simulation, functional safety, transient faults, ISO26262, RTL**

I. Introduction

During the design of ISO26262 [1] compliant chips, designers need to evaluate effectiveness of the design to deal with random hardware faults. This is usually done by the fault injection simulations. The goal of a fault injection experiment is to exercise the system's fault protection capabilities. Faults which cause the system to fail in the absence of fault detection capabilities are defined to be *critical*. A *critical fault*, if undetected in presence of fault processing mechanism, will result in a failure of the system under test. Using critical faults to estimate fault coverage eliminates the possibility of fault injection experiments to produce no errors. Several approaches to generate the critical fault list to speed up the fault injection campaigns have been proposed. However, to the best of the authors' knowledge this is the first work where dynamic HDL slicing has been implemented in order to minimize the number of fault injections. The main contributions proposed by this work as follows:

- Dynamic slicing on HDL for critical fault list generation.
- Language-agnostic RTL fault injection.

As a result, significant speed-up of the fault injection simulation is achieved. Experimental results show that the proposed methodology doubles the simulation speed when comparing to the state-of-the-art optimizations based on static cone approach. Only fault model implemented in this paper is based on single-clock-cycle bit-flip faults within the RTL registers. This fault model is targeting single Single-Event-Upsets (SEUs) in all the flip-flops of the design. The proposed methodology is demonstrated on Cadence tools but it remains applicable to other tool flows as well.

This research was supported by project RESCUE funded from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

In the majority of the published literature [2], [3] fault location and fault insertion time are randomly selected as opposed to the methodology explained in this paper. In addition, previous works [4] have demonstrated that with randomly selected fault lists the ratio of faults which do not produce errors may range as low as 2 to 8 per cent, depending on the system under simulation. Therefore, minimization of fault injection locations has a potential to reduce the time of the fault injection campaign significantly while allowing injection and simulation of a considerably larger number of relevant faults. Additionally, dynamic slicing technique is used in [5], [6] for statistical bug localization in RTL. Different from the works listed above, this paper proposes a dynamic HDL slicing based technique that implicitly covers the golden run fault collapsing, thereby significantly speeding up the fault injection process.

II. Proposed Methodology

The proposed methodology is outlined in Fig. 1. We explain the details of the methodology in the following paragraphs by using a motivational example depicted in Fig. 2.

Static slice(1) shows the dependency between HDL statements [7]. Static slice column in Fig. 2 shows the HDL statements which are in static slice of *TAR_F* output. Fig. 2 also implies that, static slice does not depend on clock cycles (shown as C1, C2, C3, C4 and C5). In this work, Cadence® JasperGold Formal Verification Platform is used to calculate backward static slice.

In parallel to static slicing step, the RTL design is simulated in Cadence® Xcelium™ simulator to dump and analyse coverage data(2). In this step, we dump coverage data for each clock cycle so that we can find what statements in the RTL are executed for each clock cycle. In the proposed methodology, one clock cycle defines the size of our dynamic slice. We use code coverage which measures how thoroughly a testbench exercises the lines of HDL code. At the end of this step, we

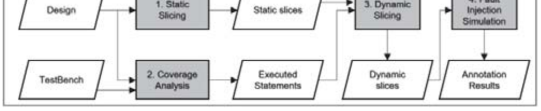


Fig. 1. Proposed Dynamic HDL slicing based fault injection methodology.

generate executed statements to use it in the next step. Fig. 2 shows executed statements for five clock cycles (C1, C2, C3, C4, C5).

Dynamic slicing(3), as it is implemented here, includes those statements that actually affect the value of a variable for a particular set of inputs of the RTL so it is computed on a given input [8]. It provides more narrow slices than static slice and consists of only the statements that effect the value of a variable for a given input. In a nutshell, dynamic slice is the intersection of static slice and executed statements as in the Fig. 2. For instance, during the time window C5, register *FF* (Line 27) is not in dynamic slice meaning that we do not need to inject fault in *FF* at C5 time window. Dynamic slice gives us critical faults and eliminates those faults that are not critical. In this way, we manage to reduce fault list by injecting only critical faults. This provides significant speed-up in the fault injection simulation time as each injected fault increases total run time of fault injection campaign.

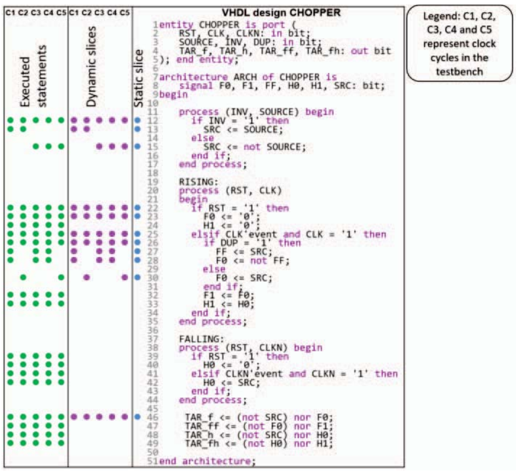
For the fault injection simulation step(4), we use Cadence® Xcelium™ Fault Simulator. Fault injection simulation selects critical faults from the dynamic slices, injects them at the specified time and evaluates the fault propagation.

III. Experimental Results

In order to verify the accuracy of proposed fault injection method, we firstly integrate our methodology into Cadence flow, then we execute our application on different designs that are available in [9] and [10]. Table I shows the details for both static slice which is state-of-the-art approach and dynamic slice optimization. For the smaller *chopper* example, total CPU time of overall regression is reduced to 1.2s when compared to static slice optimizations. For the more complex *simple_spi* design, two-dimensional memory is selected as a fault target. As a result, we reduce the fault list to the critical faults and achieve 11.2 times shorter CPU time in dynamic slice optimization.

IV. Conclusions

This paper proposes a methodology to optimize fault injection campaigns by pruning the fault list to the critical faults identified using a dynamic HDL slicing technique that provides for fault list collapsing. In this way, we narrow down the fault space and reduce execution time of fault injection simulation campaigns. Experimental results show that we achieve significant speed-up of the fault injection simulation when comparing to the state-of-the-art flows.



```
 1entity CHOPPER is port (
 2    RST, CLK, CLKN: in bit;
 3    SOURCE, INV, DUP: in bit;
 4    TAR_f, TAR_h, TAR_ff, TAR_fh: out bit
 5); end entity;
 6
 7architecture ARCH of CHOPPER is
 8    signal F0, F1, FF, H0, H1, SRC: bit;
 9begin
10
11    process (INV, SOURCE) begin
12      if INV = '1' then
13        SRC <= SOURCE;
14      else
15        SRC <= not SOURCE;
16      end if;
17    end process;
18
19    RISING:
20    process (RST, CLK)
21    begin
22      if RST = '1' then
23        F0 <= '0';
24        H1 <= '0';
25      elsif CLK'event and CLK = '1' then
26        if DUP = '1' then
27          FF <= SRC;
28          F0 <= not FF;
29        else
30          F0 <= SRC;
31        end if;
32        F1 <= F0;
33        H1 <= H0;
34      end if;
35    end process;
36
37    FALLING:
38    process (RST, CLKN) begin
39      if RST = '1' then
40        H0 <= '0';
41      elsif CLKN'event and CLKN = '1' then
42        H0 <= SRC;
43      end if;
44    end process;
45
46     TAR_f <= (not SRC) nor F0;
47     TAR_ff <= (not F0) nor F1;
48     TAR_h <= (not SRC) nor H0;
49     TAR_fh <= (not H0) nor H1;
50
51end architecture;
```

Fig. 2. HDL slicing on a motivational example chopper [9].

References

[1] I. S. Organization, "Iso 26262 - road vehicles - functional safety," *International Organization for Standardization*, 2011.

[2] X. Iturbe, B. Venu, and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu," in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Sept 2016, pp. 91–96.

[3] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra, "Processor core profiling for seu effect analysis," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, March 2018, pp. 1–6.

[4] J. Raik, U. Repinski, M. Jenihhin, and A. Chepurov, "High-level decision diagram simulation for diagnosis and soft-error analysis," *Design and Test Technology for Dependable Systems-on-Chip*, pp. 294–309, 2011.

[5] M. Jenihhin, A. Tšepurov, V. Tihhomirov, J. Raik, H. Hantson, R. Ubar, G. Bartsch, J. H. M. Escobar, and H. Wuttke, "Automated design error localization in rtl designs," *IEEE Design Test*, vol. 31, no. 1, pp. 83–92, Feb 2014.

[6] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. D. Guglielmo, G. Pravadelli, and F. Fummi, "Combining dynamic slicing and mutation operators for esl correction," in *2012 17th IEEE European Test Symposium (ETS)*, May 2012, pp. 1–6.

[7] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura, "Program slicing on vhdl descriptions and its applications," 1996.

[8] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(88)90054-3

[9] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for vhdl," *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 125–137, 2002.

[10] (2018) Opencores. [Online]. Available: http://www.opencores.org

TABLE I
Fault Injection Campaign Results for Chopper and Simple_SPI Designs

Design Name	**chopper**		**simple_spi**	
Optimization type	**Static Slice**	**Dynamic Slice**	**Static Slice**	**Dynamic Slice**
Observation list	tar_f		dat_o	
Fault target	F0, FF	dynamic slices	mem[][]	dynamic slices
Total number of injected faults	410	255	210080	960
Number of detected faults	220	137	1696	609
Number of undetected faults	190	118	208384	351
Total CPU time of overall regression	1.33s	1.2s	171.5s	15.2s

Appendix 5

V

Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Said Hamdioui, and Christian Sauer. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 129–1295, 2019

Combining Fault Analysis Technologies for ISO26262 Functional Safety Verification

Felipe Augusto da Silva[1,2], Ahmet Cagri Bagbaba[1], Said Hamdioui[2] and Christian Sauer[1]
[1]Cadence Design Systems, Feldkirchen, Germany - {dasilva, abagbaba, sauerc}@cadence.com
[2]Delft University of Technology, Delft, The Netherlands - {f.augustodasilva, s.hamdioui}@tudelft.nl

Abstract— **The development of Integrated Circuits for the Automotive sector imposes on complex challenges. ISO26262 Functional Safety requirements entail extensive Fault Injection campaigns and complex analysis for the evaluation of deployed Software Tools. This paper proposes a methodology to improve Fault Analysis Tools Confidence Level (TCL) by detecting errors in the classification of faults. By combining the strengths of Automatic Test Pattern Generators (ATPG), Formal Methods and Fault Injection Simulators we are able to automatically generate a Test Environment that enables the validation of the tools and provides supplementary information about the design behavior. Our results showed fault detection rates above 99% including information to improve ISO26262 metrics calculation.**

Keywords - ISO26262; Fault Injection; Formal Methods; Simulation; Tool Confidence Level; Functional Safety; Verification; ATPG.

I. INTRODUCTION

Functional Safety Verification is one of the most challenging steps for Integrated Circuit (IC) compliance with ISO26262. Particularly for safety-critical applications such as autonomous driving, where in case of a failure, a life-threatening situation can happen. For such applications, the system must include Safety Mechanisms being able to detect up to 99% of the random faults susceptive of the design. At the IC Gate-Level representation, the number of faults can easily reach the millions figure, requiring huge efforts to analyze all of them. In addition, ISO26262 requires that all possible malfunctions of tools (used during fault analysis) have to be considered, meaning that developers have to assess the level of confidence on the outputs of a tool. The tool may require compliance with Tool Qualification requirements; this even increases the complexity of functional safety verification. Therefore, there is a high demand for effective Functional Safety Verification methodologies allowing the reduction of costs while maintaining the same levels of safety.

The commonly used method for Functional Safety Verification is Fault Injection (FI) Simulation [1][2][3][4]. The purpose is to show that fault effects can propagate to outputs and that Safety Mechanisms can detect them. Propagation of faults during simulation is key for achieving ISO26262 requirements. An injected fault that is not observed on the outputs, must be re-simulated or proven to be untestable. In order to provoke propagation of all faults, complex verification environments with numerous test inputs are required, resulting in long FI Campaigns. To address this challenge, we can deploy different verification technologies in a single methodology. Formal Methods can be employed to leverage the most appropriate setups for simulation campaigns. The ability of formal in analyzing design behavior to all test inputs can help to identify untestable faults and to determine test inputs for corner cases [5][6][7]. Anyhow, Formal Methods are not capable of analyzing all faults in an acceptable time frame. Therefore, another solution is still required to analyze a large portion of the faults. The application of automatically generated ATPG Testbenches can decrease the efforts on the development of simulation environments. ATPG tools are able to create test patterns that potentialize fault propagation. Simulation can be performed with the generated test vectors aiming to achieve better failure coverage with reduced simulation times [8][9]. Nonetheless, ATPG focuses on manufacturing test and is not optimal for determining untestable faults or covering faults on areas out of the scan chains reach. Even though Simulation, Formal Methods, and ATPG have complementary strengths, to the best of our knowledge, they were not previously combined in a single fault analysis flow that aims at fault propagation for compliance to ISO26262 requirements.

Our work takes advantage of three different technologies aiming to verify the correctness of fault classification while providing data to support traditional FI Campaigns. Initially, ATPG is used to generate a verification environment that provides high fault propagation rate. The outputs from ATPG are used by the FI Simulator, to verify the functional behavior of the design under each fault. In parallel, Formal Methods are applied to identify faults that are untestable and determine the behavior of faults that are not covered by ATPG. Finally, the outputs of each tool are verified against each other to identify malfunctions, increasing the confidence in the tool's outputs, as required by ISO26262 [10]. The main contributions of our methodology are:

- Increasing Tool Confidence Level according to ISO26262. By providing an automated flow for error detection in Fault Analysis tools, we can avoid the extensive ISO26262 Tool Qualification requirements.
- Identification of untestable faults. Formal Methods can prove that faults cannot be tested, and therefore can be ignored during safety metrics calculation, increasing compliance with ISO26262 fault metrics.

- Initial assessment of the fault propagation behavior by the deployment of ATPG Test Environments and Formal results. The achieved fault detection rates, above 99% on tested designs, can be employed to support the ISO26262 Functional Safety Verification.

This paper is organized as follows. Section II investigates how fault analysis is implemented by different technologies. Section III describes the proposed methodology. Section IV presents the validation process and explain our results. And last, Section V presents our final conclusions.

II. FAULT ANALYSIS

This section investigates how fault analysis is implemented by different technologies. The examination aims to identify the strengths and weaknesses of each solution and determine how they comply with Functional Safety requirements. ISO26262 requires that any component that implements a safety-related functionality, reach a minimum level of tolerance to random hardware failures. Coverage for this type of failure is usually increased by the addition of Safety Mechanisms to the design. Safety Mechanisms, as defined by ISO26262, should be able to detect faults or control failures in order to achieve or maintain a safe state.

The effectiveness of the design to cope with random hardware failures should be quantitatively demonstrated by the calculation of metrics defined by the standard [11]. It is necessary to evaluate the efficiency of the Safety Mechanisms to handle critical faults, contributing to achieving targeted safety metrics. Fault Injection Simulation is a widely used technique to perform this analysis being the method recommended by ISO26262.

A. Fault Injection Simulation

Analysis of Fault Injection by Simulation is widely used and available in a variety of tools. These tools are able to analyze a Register Transfer Level (RTL) or Gate-Level (GTL) descriptions of an IC and, based on given test inputs, simulate their behavior. The effect that a fault produces in the design is determined by comparing the behavior of the design with and without faults. The flow implemented by Fault Injection Simulation Tools is described below:

1) Elaboration of RTL/GTL design description.
2) Fault List Generation: candidates for fault injection are defined for each available fault model. The user should define rules (e.g. all signals) to identify fault node candidates and fault models (e.g. Stuck-at-0 (SA0) and Stuck-at-1 (SA1)). Information is stored in a fault database.
3) Fault List Optimization: Faults list is analyzed to identify candidates for optimization. Based on the elaboration results, tools can estimate the behavior of some faults decreasing the number of faults to be simulated. Information is updated on the fault database.
4) Good Simulation: fault-free behavior of design is simulated. The user should define observation points in the design to identify: (1) Fault propagation to a functional output: functional strobes; (2) Activation of the Safety Mechanism: checker strobes. The values of the Strobes during good simulation are stored.
5) Fault Injection Simulation: For each fault in the fault database, the design faulty behavior is simulated, and the observation points compared against the reference values from the Good Simulation. The behavior of the design under each fault is analyzed and stored.

FI Simulation determines the behavior change provoked by a fault when the effect is observable in one of the outputs (strobes). Faults that don't produce changes in the strobes are classified as Undetected. This is considered a weak result of the simulation, as a different test may cause fault propagation. Fault propagation is required to assure correct classification. If there are no test stimulus that provokes the propagation of a fault, this should be proved by analysis. For that reason, FI Simulation demands the development of complex Testbenches and additional untestable fault analysis.

B. Formal Methods

Identification of untestable faults requires proof that the fault cannot be tested by ANY functional test stimulus. Formal analysis appears as a good alternative for this purpose since it is not limited to a specific time or state. Instead, the scope is global, and every evaluation context and test stimulus is considered. Consequently, formal analysis can exhaustively prove that a fault can never produce any failure. This class of faults can be considered untestable and don't require further fault simulation.

Different EDA vendors explore fault analysis capabilities in their formal solutions. Generally speaking, these solutions automatically generate properties, not requiring knowledge of formal languages. In addition, they allow integration with FI Simulators providing fault lists optimization and reducing simulation campaigns. Tools used for fault formal analysis usually apply two main fault analysis techniques, Standard Analysis, and Advanced Analysis.

The Standard Analysis aims to determine the testability of faults. It is applied as a pre-qualification flow for simulation, to reduce the fault list by identifying untestable faults. The testability of the faults is determined by verifying:

- if there is a physical connection between the fault location and the observation points (strobes).
- if the signals that drive the fault node allows activation of the fault.
- if the fault could be observable in at least one strobe of the design.

A fault that does not pass these verifications can be classified as untestable. In addition, the fault list may be optimized by Fault Relation Analysis. The tool analyzes the design to determine the relationship between fault pairs. Fault pairs are then included in the same Collapsing Group. The behavior of all Collapsing Group is predicted by simulation of only one representative of the group, called the Prime Fault.

The Advanced Analysis deploys formal techniques to analyze propagation and activation of the faults. Activation

TABLE I
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

Analysis indicates whether the fault can be functionally activated from any combination of inputs. Propagation Analysis verifies if there is a combination of inputs that provoke fault propagation. Advanced Analysis will classify the faults, which were not previously classified by the Standard Analysis, in three groups:

- Untestable: Faults that cannot be activated or propagated.
- Dangerous: The tool identified a combination of test inputs that results in fault propagation.
- Unknown: All the others.

Formal properties to perform the Advanced Analysis are automatically generated and verified with respect to all possible input stimulus. The Advanced Analysis relies on formal properties and analysis to prove the properties to be true. The analysis of formal properties is time-consuming and cannot find results for all faults in complex designs. For that reason, this analysis is often applied as a last resource, on the faults that were not classified after fault injection simulation.

The different strengths of Simulation and Formal can complement each other. An integrated fault analysis flow allows the deployment of the Standard Analysis before the start of the simulation. The analysis will reduce the number of faults to be simulated by leveraging results for untestable faults and collapsing groups. After the simulation, Advanced Analysis can be executed on the remaining undetected faults to verify if there is a combination of test inputs that would result in fault propagation.

Even with the combination of Formal and Simulation, the development of the test environments is challenging. Advanced Analysis from Formal tools, that can support the identification of test stimulus for fault propagation, are time-consuming and cannot find results for all fault list. In this context, ATPG appears as a possible solution for generating Testbenches and test inputs that can be used for the FI Simulation.

C. Automatic Test Pattern Generator

Test patterns can be generated 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 is detected if any of the outputs are different from the expected pattern. Nowadays, ATPG is a well-established technology being used on the development of almost all IC. ATPG tools can generate a minimal group of test vectors to achieve acceptable levels of manufacturing defects detection. In addition, tools can generate reports about the testability of each defect, allowing the generation of metrics to indicate test quality and test application time.

Usually, an ATPG flow receives as inputs a Gate-Level description of an IC and specification of the scan chains. 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 outputs are programmed in a Test Equipment that will be used in IC manufacturing tests. The Test Equipment applies the test patterns in the inputs of the circuit and monitors the outputs to verify if the values are the expected ones. We propose a similar approach using FI Simulation. Instead of using a Test Equipment, we apply the ATPG test patterns on the design simulation and use the strobe functionality to monitor the outputs of the design. During the Good 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. This way, we can use the propagation capabilities of ATPG to identify behavioral changes caused by injected faults.

The fault propagation potential of ATPG test environments is a powerful benefit for compliance with Functional Safety. However, ATPG focuses on manufacturing tests and the estimated results should be demonstrated via simulation. In addition, ATPG doesn't consider untestable faults and faults out of the scan chain reach. Formal Analysis can be deployed for addressing these cases.

Table I summarizes the strengths and weakness of each technology. Considering this examination, we propose a methodology that highlights the strengths of Simulation, Formal and ATPG for Functional Safety Verification.

III. PROPOSED METHODOLOGY

This section describes the application of three fault analysis technologies in an efficient methodology for ISO26262

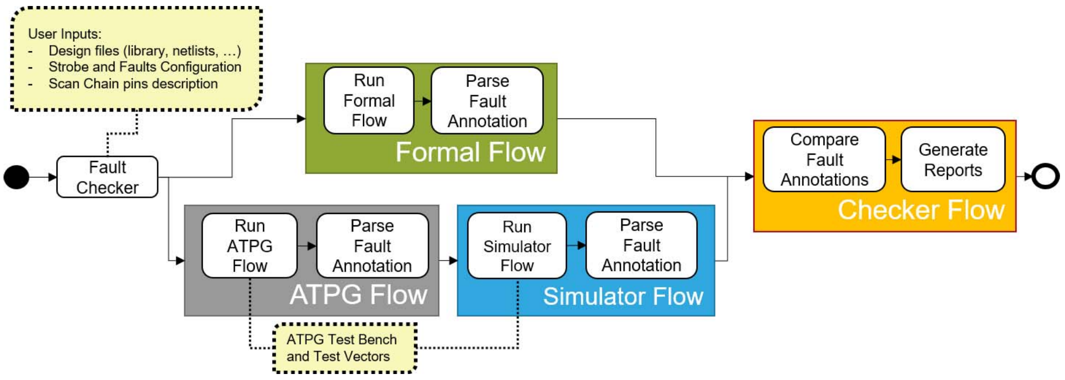


Fig. 1. Fault Checker Execution Flow.

Functional Safety Verification. The methodology highlights the strengths of Simulation, Formal and ATPG to generate a comprehensive fault analysis report. An application was developed aiming to automate the execution of the different tools. The Fault Checker application implements a generic control flow that is configurable with tools from different vendors. In the end, the reports of each tool are parsed and saved in a common format. The fault classification of each tool is combined in a final report that allows the identification of tool malfunctions and detailed analysis of faults behavior.

The Fault Checker application must be configured with scripts to control the execution of each tool and with the rules for parsing the reports. Also, the user must provide design-specific information, as fault targets and observation points (strobes). With all the required information, the application can start the execution of the ATPG and Formal flows. As these two flows are independent, they can be executed in parallel using different CPUs. Simulator flow requires the ATPG Testbench and test vectors to start. So, after the ATPG flow is finished, the Fault Checker will extract the generated Test Environment and will use it for the FI Simulation. At the end of each flow, the reports generated by the tools are parsed to a common format and saved. Finally, at the end of all flows, the relevant parsed data is retrieved and compared. The comparison is based on rules that associate the classifications used by each tool. In case a rule is not obeyed, the Fault Checker will include a Warning tag, informing that this fault requires attention from the designer. Fig. 1, illustrates the execution flow of the Fault Checker application.

Results can be analyzed in a CSV report that details the classification of each fault by each tool. An error caused by a malfunction in one of the tools will be indicated by a Warning in the report. For example, if the Simulator classifies a fault as Detected and Formal classifies the same fault as Safe, this would indicate a malfunction in one of the tools. A sample of the detailed report is demonstrated in Table II.

In addition to malfunction indication, the report provides supplementary information for fault analysis. For example, signal "dut.u0.sig2" in Table II, is classified as Undetected by the Simulator and Ignored by ATPG. However, 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. This information can be used on a new FI Simulation 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 classified as Undetected and Ignored. Results from the formal analysis can be used to demonstrate that the fault cannot propagate to a strobe, and therefore can be considered untestable, contributing to achieving ISO26262 metrics. Any other discrepancy between the faults is indicated in the report, as illustrated by signal "dut.u0.INsT0.0".

IV. Validation

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

A. *Validation Setup*

The methodology was validated by deploying the Fault Checker application on example designs. First, the Fault Checker must be configured with the tools to execute each flow. Our work has adopted Cadence® Xcelium^{TM} Fault

TABLE II
Fault Checker Report Example.

Signal Name	Fault Type	Formal Classification	Simulator 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

TABLE III
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

Simulator (XFS), Cadence® JasperGold (JG) Formal Verification Platform Functional Safety Verification (FSV) and Cadence® Modus DFT Software Solution ATPG component, as the representatives of each technology.

The selection of the designs contemplated different levels of complexity and the availability of Functional Testbenches. Complexity was determined by the number of fault targets in each design. The ISO26262 defines that all cell ports in the IC Gate-Level representation should be analyzed for faults. The selected designs were synthesized using the standard cell reference libraries provided with Cadence 45nm Generic Process Design Kit (GPDK) [12]. The selected designs are available on the IWLS 2005 benchmark [13]. 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.

Designs (1) and (2) were initially deployed to verify that the Fault Checker application was working 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. The other designs were deployed to verify the behavior of the Fault Checker application when analyzing larger designs. In addition, for designs (3) and (4), the achieved results were compared with fault injection results using Functional Testbenches only. The achieved results are described in the following sections.

The experiments were executed on two Intel Xeon E5-2680 CPUs with 16 Cores and 252 GB of memory each. Being the Formal flow executed on CPU1 and ATPG followed by Simulation Flow in CPU2. Parallel fault injection simulations were performed to improve the overall time of the Simulation Flow.

B. Results

Table III demonstrates the results of the methodology for the selected designs. It details, for each design, the total number of faults, the fault detection rate, and the Pass/Warning indication resulting from the Fault Checker verification.

During the Up Down Counter design verification, the Fault Checker confirmed that all faults have equivalent classifications. As the example is relatively simple, the different technologies can determine that all faults can propagate to observation points (strobes).

TABLE IV
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%

For the Memories design, the application detected 6 faults with discrepant classifications. In this example, the Warnings were due to classifications of Safe Faults by Formal and Undetected by the Simulator. For these 6 faults, the Formal analysis proves that the faults are untestable, and can be disregarded, improving results for ISO26262 metrics calculation.

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 can be declared as untestable; 23 were classified as Dangerous by Formal and Undetected by the Simulator, meaning that these faults can be Detected in Simulation by applying the results from Formal as test inputs; 46 faults were considered Undetected by Simulation and ATPG and Unknown by Formal, indicating that none of the tools was able to define the possible behavior of these faults, and they require manual analysis; 6 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, 7 faults were classified as Dangerous by Formal and Undetected by 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. As the classifications are not conclusive, these faults should be manually analyzed.

To analyze the capability of the methodology for fault classification, we compared the Fault Checker results with results from fault injection when using a Functional Testbenches. The AC97 and Conmax designs include simulation environments for verification of their functionalities. Table IV demonstrate results of the FI simulation of the AC97 and the Conmax designs when deploying the Functional Testbenches and when using the Fault Checker. Due to the characteristics of fault propagation provided by the ATPG Testbenches, 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.

The Undetected classification is inconclusive for fault analysis. Undetected faults must be proven Untestable to collaborate to ISO26262 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. If we have applied Functional Testbenches to achieve the same level of fault detection from the Fault Checker, we would need to repeat the Fault Injection Campaign with new test inputs, until all faults get propagated to outputs, demanding the development of new Test Environments and longer FI Campaigns.

C. 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 a high degree of confidence in tool error detection, resulting in a Tool Confidence Level (TCL) of one. A methodology with TCL1 doesn't require Tool Qualification, avoiding big efforts on documentation and analysis for compliance with ISO26262 [10]. Second, identification of Safe faults collaborates with ISO26262 compliance. By proving that a fault is untestable, we are able to disregard it, decreasing the total number of faults to be simulated and improving ISO26262 metrics [11]. Third, the proposed methodology achieved substantial fault detection rates. The use of ATPG test vectors during simulation and identification of Dangerous faults by Formal, provide extra information about the design behavior. In summary, our results can be applied to support the following aspects of ISO26262 Functional Safety Verification:

- Avoid efforts with Tool Qualification by automating tool error detection.
- Identification of Untestable Faults allows improvement of ISO26262 metrics and reduction of the number of faults to be simulated.
- Fault supplementary data can be used to support further fault injection campaigns.

Even though we have achieved high fault detection rates, we need to consider that the examples used were of average complexity. One of the next steps of our work is to apply our methodology to more complex designs. We need to explore how the fault detection provided by ATPG in complex designs can leverage the Safe and Dangerous classifications from Formal for the achievement of ISO26262 requirements.

Another aspect to acknowledge is the possibility of changes in the fault propagation patterns when ATPG scan chains are disabled. The application of our technique in more complex designs, for instance, an Automotive CPU, should consider this effect and employ formal results to assess differences in the classification of the faults.

V. CONCLUSIONS

Due to the harsh requirements for random hardware failures tolerance, Functional Safety verification is a challenging step for ISO26262 compliance. Fault analysis, as part of this process, becomes a extensive procedure, that is usually repeated numerous times until the metrics for fault detection are achieved. Furthermore, ISO26262 requires specific criteria to determine the level of confidence in the adopted software tool, increasing the efforts even further. We propose a methodology that deploys ATPG and Formal to support Simulation results and to decrease the overall efforts of ISO26262 compliance. Our methodology enables the use of test environments created with ATPG tools for the simulation of faults, and the use of Formal for identification of untestable faults. Formal results allow the optimization of the Fault List, reducing the number of faults to be simulated, and the generation of test vectors for the detection of corner cases. In addition, the results of the tools are compared to identify potential malfunctions. The inclusion of redundancy as a method to detect malfunctions in tools is a suggested method for achieving ISO26262 Tool Confidence [10]. Our results have shown high fault detection rates, achieving more than 99% of detected faults. In addition, detailed fault information provided contributes to achieving ISO26262 metrics.

ACKNOWLEDGMENT

This research was supported by project RESCUE funded from the European Unions Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

REFERENCES

[1] 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.

[2] S. Pateras and T.-P. Tai, "Automotive semiconductor test," in *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, apr 2017.

[3] 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.

[4] 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.

[5] K. Devarajegowda and J. Vliegen, "Deploying formal and simulation in mutual-exclusive manner using jaspergolds proofcore technology," in *Cadence User Conference CDNLive EMEA*, 2017.

[6] S. Marchese and J. Grosse, "Formal fault propagation analysis that scales to modern automotive SoCs," in *2017 Design and Verification Conference and Exhibition DVCON Europe*, 2017.

[7] A. Traskov, T. Ehrenberg, and S. Loitz, "Fault proof: Using formal techniques for safety verification and fault analysis," in *2016 Design and Verification Conference and Exhibition DVCON Europe*. DVCON, 2016, pp. 27–32.

[8] 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.

[9] 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.

[10] ISO, *ISO 26262 - Road Veichles - Functional Safety - Part 8: Supporting processes*, International Standardization Organization Std., Nov. 2011.

[11] ISO, *ISO 26262 - Road Veichles - Functional Safety - Part 5: Product development at the hardware level*, International Standardization Organization Std., Nov. 2011.

[12] *GPDK045 Reference Manual*, Revision 5.0 ed., Cadence Design Systems , Inc., 2016.

[13] C. R. Berkeley, "International workshop on logic and synthesis (IWLS) 2005 benchmarks," Tech. Rep., 2005.

Appendix 6

VI

Ahmet Cagri Bagbaba, Maksim Jenihhin, Jaan Raik, and Christian Sauer. Accelerating transient fault injection campaigns by using dynamic hdl slicing. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019

Accelerating Transient Fault Injection Campaigns by using Dynamic HDL Slicing

Ahmet Cagri Bagbaba*†, Maksim Jenihhin†, Jaan Raik†, Christian Sauer*
*Cadence Design Systems, Munich, Germany; † Tallinn University of Technology, Tallinn, Estonia
Email: *{abagbaba, sauerc}@cadence.com, †{maksim.jenihhin, jaan.raik}@taltech.ee

***Abstract*—Along with the complexity of electronic systems for safety-critical applications, the cost of safety mechanisms evaluation by fault injection simulation is rapidly going up. To reduce these efforts, we propose a fault injection methodology where Hardware Description Language (HDL) code slicing is exploited to accelerate transient fault injection campaigns by pruning fault lists and reducing the number of the injections. In particular, the dynamic HDL slicing technique provides for a critical fault list and allows avoiding injections at non-critical time-steps. Experimental results on an industrial core show that the proposed methodology can successfully reduce the number of injections by up to 10 percent and speed-up the fault injection campaigns.**

***Index Terms*—fault injection, fault simulation, functional safety, transient faults, ISO26262, RTL, CPU**

I. Introduction

With new and increased capabilities in applications such as autonomous driving, the complexity of electronics systems for safety critical applications is growing exponentially. This is causing a shift in the traditional design flow and is pushing ISO26262 compliance down in the semiconductor chain to the individual IP provider and even into the traditional Electronic Design Automation tools. As a result, functional safety compliance becomes a part of the requirements for the development of complex electronics systems. During the design of ISO26262 compliant chips, designers need to evaluate effectiveness of the design to deal with random hardware failures. This is usually done by Fault Injection Simulations. Also, ISO26262 standard highly recommends using of fault injection during the development process of integrated circuits [1].

Fault injection is a powerful technique that shows the behaviour of a circuit under the effect of a fault [2]. The objective of fault injection is to mimic the effects of faults originating inside a chip as well as those affecting external buses. Different approaches to fault injection and dependability evaluation have been proposed. These include emulation-based fault injection using FPGA architectures as hardware accelerators to speed up estimation of systems' fault tolerance [3], [4] and formal method based approaches [5], [6]. This paper focuses on the simulation-based fault injection approach, which can be applied to larger designs compared to the formal and emulation-based solutions.

Having enormous number of possible faults in modern designs is a major drawback of simulation-based fault injection technique as designers need to execute a fault-free simulation as well as thousands of faulty simulations [7]. Therefore, it is too hard to inject all possible faults in an acceptable time in all possible locations and at each clock cycle [8]. One solution is to use Statistical Fault Injection (SFI) [8] in which only a randomly selected subset of possible faults is injected. SFI can provide a better execution time by reducing the number of the injections with an error margin. Moreover, [9] have demonstrated that with randomly selected fault lists the ratio of faults which do not produce errors may range as low as 2 to 8 percent, depending on the design under simulation. In consequence, minimization of fault injection locations or pruning fault lists are advantageous ways to reduce the fault injection simulation time significantly while allowing injection of a considerably larger number of relevant faults.

This work proposes a simulation-based fault injection methodology based on Dynamic HDL Slicing to minimize the number of fault injections. The proposed methodology identifies critical faults which cause the system to fail in the absence of a safety mechanism, and injects only critical faults during the transient fault injection simulation campaigns. Using critical faults to estimate fault coverage eliminates the possibility of fault injection experiments to produce no error. The main contribution of this work is three-fold as follows:

- Dynamic slicing on HDL to generate critical fault list
- Implicit fault collapsing within the slicing model: The fault list obtained by the proposed slicing method has an additional feature of avoiding injections at time-steps as data inside registers is not being consumed.
- Language-agnostic RTL fault injection supported by industrial grade EDA tool flow

As a result, this method can successfully reduce the number of fault injections on an industrial core. The fault model implemented in this paper is based on single-clock-cycle bit-flip faults within the RTL registers. This fault model is targeting single Single-Event-Upsets (SEUs) in all the registers of the design. The proposed methodology is demonstrated on Cadence tools but it remains applicable to other tool flows as well. This work is an extension of our previous work [10]. The major difference is that we extend dynamic slices' scope by including both sequential and combinational parts as it is explained in Section III-C1. This brings 100% accuracy in the results. In the previous work, less accuracy is adopted as only sequential parts are considered in dynamic slices. The

978-1-7281-2769-9/19/$31.00 ©2019 IEEE

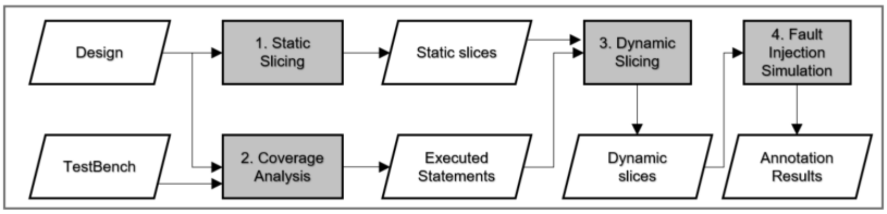


Fig. 1. Proposed HDL slicing based fault injection methodology [10].

second difference is that we evaluate our methodology in the industrial size CPU with different workloads to demonstrate the potential of the proposed methodology.

This paper is structured as following. Section II gives an overview of related works. We describe our dynamic HDL slicing methodology in Section III. Experimental results are shown in Section IV. Section V concludes this paper.

II. Related Works

There exist many advanced tools and methods for simulation-based fault injection. In [11], a tool called VERIFY (VHDL-based Evaluation of Reliability by Injection Faults Efficiently) is presented that utilizes an extension of VHDL for describing faults correlated to a component, enabling hardware manufacturers, which provide the design libraries, to express their knowledge of the fault behaviour of their components. Although it provides multi-threaded fault injection as well as checkpoints and comparison with a golden run to speed up the simulation of faulty runs, the drawback is that it requires modification of the VHDL language itself. [12] proposes MEFISTO-C: A VHDL-based fault injection tool that conducts fault injection experiments using VHDL simulation models. A variety of predefined fault models are supported by the tool; however, it does not provide specific optimizations to speed up the simulation.

Several approaches to generate the critical fault list to be considered as the basis of fault list injection have been proposed. In [13], a method for generating a critical fault list is presented. The system under test is described by a data flow graph, the fault tree is constructed by applying the instruction set architecture fault model to the data flow description with a reverse implication technique, the fault injection is performed, and fault collapsing on the fault tree is employed. The proposed method is very costly in terms of CPU time and it therefore not applicable to systems with high complexity.

[7] presents a new technique and a platform for accelerating and speeding-up simulation-based fault injection in VHDL descriptions. Use check-pointing to reload the fault-free state if the design allowing to start the fault simulation from the clock-cycle of fault injection. In addition, a golden-run fault collapsing technique is utilized that discards all fault injections between read-write and write-write operations of the memory elements. However, the approach does not take advantage of the dynamic slicing benefits. [9] proposes fault collapsing based on extracting high-level decision diagrams from the VHDL model. Although significant speed-up can be achieved, the step of efficient decision diagram synthesis from the full synthesizable subset of VHDL remains an issue.

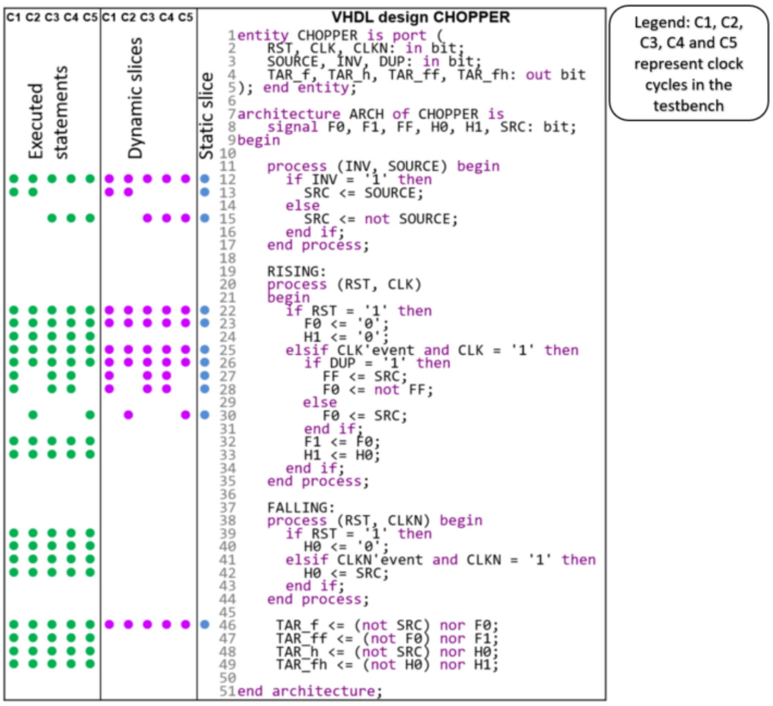
There are several papers dealing with transient fault injection. [14] shows the results collected in a series of fault injection experiments conducted on a commercial processor. Here, the authors inject a fault in a given sequential element at a given instant of time. However, as it is hard to inject a fault in each of the tens of thousands sequential elements in the processor, the execution is divided into the parts and, for each of these parts, a random fault injection instant is selected. [15] analyses fault injection campaign in the CPU registers by choosing a random instant when the fault is injected. [16] identifies the optimal set of flip-flops but injection time is randomized uniformly over the active region of the simulation. Similarly, [17] injects a fault randomly in time and location in RT-level. Lastly, [18] deals with single and multiple errors in processors by randomly selecting injecting time and choosing registers. As opposed to these works, our approach shows the fault injection time explicitly instead of random instants.

Dynamic slicing technique is used in [19], [20]. The former uses dynamic slicing for statistical bug localization in RTL. The latter proposes dynamic slicing and location-ranking-based method for accurately pinpointing the error locations combined with a dedicated set of mutation operators.

Different from the works listed above, this paper proposes a dynamic HDL slicing based technique that implicitly covers the golden run fault collapsing, thereby significantly speeding up the fault injection process.

III. Fault Injection based on Dynamic HDL Slicing Technique

In this work, fault injection simulation campaigns are optimized by pruning the fault list to the critical faults identified using HDL slicing on the RTL design model. The proposed flow is shown in Fig. 1 and starts with the (1) extraction of static slices for the target observation point. In parallel, code coverage data is generated by (2) simulation-based code coverage analysis for the design with pre-defined stimuli in the testbench. Next, (3) the dynamic slicing procedure identifies the intersection of the identified static slice and covered code items and results in a set of clock-cycle-long dynamic



```
 1entity CHOPPER is port (
 2   RST, CLK, CLKN: in bit;
 3   SOURCE, INV, DUP: in bit;
 4   TAR_f, TAR_h, TAR_ff, TAR_fh: out bit
 5); end entity;
 6
 7architecture ARCH of CHOPPER is
 8   signal F0, F1, FF, H0, H1, SRC: bit;
 9begin
10
11   process (INV, SOURCE) begin
12     if INV = '1' then
13       SRC <= SOURCE;
14     else
15       SRC <= not SOURCE;
16     end if;
17   end process;
18
19   RISING:
20   process (RST, CLK)
21   begin
22     if RST = '1' then
23       F0 <= '0';
24       H1 <= '0';
25     elsif CLK'event and CLK = '1' then
26       if DUP = '1' then
27         FF <= SRC;
28         F0 <= not FF;
29       else
30         F0 <= SRC;
31       end if;
32       F1 <= F0;
33       H1 <= H0;
34     end if;
35   end process;
36
37   FALLING:
38   process (RST, CLKN) begin
39     if RST = '1' then
40       H0 <= '0';
41     elsif CLKN'event and CLKN = '1' then
42       H0 <= SRC;
43     end if;
44   end process;
45
46    TAR_f <= (not SRC) nor F0;
47    TAR_ff <= (not F0) nor F1;
48    TAR_h <= (not SRC) nor H0;
49    TAR_fh <= (not H0) nor H1;
50
51end architecture;
```

Fig. 2. HDL slicing on a motivational example chopper [10].

slices for the given observation point. Finally, (4) the fault injection simulation selects critical faults from the dynamic slices, injects them at the specified time and evaluates the fault propagation. We explain the details of the methodology in the following subsections using a motivational example depicted in Fig. 2, i.e. a VHDL implementation of a signal *chopper* design [21]. Following subsections explain each step of the proposed methodology in detail.

A. Static Slicing

Static slice, as it is implemented in the current paper, includes all statements that affect the value of a variable *v* for all possible inputs at the point of interest, e.g., at the statement *x*, in the program. In the RTL code, static slice shows the dependency between HDL statements [22]. A simple design *chopper* in Fig. 2 has four outputs representing different chops for the input signal *SOURCE* based on the design configuration by inputs *INV* and *DUP*. It is possible to perform a search backward to find dependencies in the HDL. The resulting static slice is computed for the *chopper* design's output *TAR_F* as shown in Fig. 3 by the help of formal analysis tool's structural analysis capability. The column Static Slice in Fig. 2 marks HDL statements of a static slice on the *TAR_F* output. For instance, as the static slice of *TAR_F* does not include Line 40, *H0* is counted as outside of the static slice and for a *TAR_F* output there is no need to inject fault on *H0*. Fig. 2 also implies that, static slice does not depend on clock cycles (shown as C1, C2, C3, C4 and C5) while executed statements and dynamic slice may change for each clock cycle. In summary, static slice includes statically available information only as it does not make any assumptions on inputs. Static slice is the first step of the proposed methodology to prune fault list.

B. Coverage Analysis

In parallel to static slicing step, the RTL design is simulated in the logic simulation tool to dump and analyse the coverage data. In this step, we dump coverage data for each clock cycle so that we can find what statements in the RTL are executed for each clock cycle. In the proposed methodology, one clock cycle defines the size of our dynamic slice. We use coverage tool and coverage metrics in order to find executed statements. After loading a simulation run into the coverage tool, we can analyze coverage metrics data scored in that run. In this work, we use code coverage which measures how thoroughly a testbench exercises the lines of HDL code. Code coverage includes block coverage, branch coverage, statement coverage, expression coverage, and toggle coverage. All these coverage types except toggle coverage can be used in this work. Block coverage identifies the lines of code that get executed during a simulation run. It helps us determine if the testbench executes the statements in a block. Branch coverage complements block coverage by providing more precise coverage results for reporting coverage numbers for various branches individually. Statement coverage is just a subset of block coverage and it shows execution of all the executable statements in the RTL. Expression coverage provides information on why a conditional piece of code was executed. At the end of this step, we generate executed statements data to find dynamic slices in the next step. Fig. 2 shows executed statements for five clock cycles (C1, C2, C3, C4, C5).

C. Dynamic Slicing

Dynamic slice, as it is implemented in the current paper, includes those statements that actually affect the value of a variable *v* for a particular set of inputs of the RTL so it is computed on a given input [23]. It provides more narrow slices than static slice and consists of only the statements that affect the value of a variable for a given input.

In a nutshell, dynamic slice is the intersection of static slice and executed statements. We illustrate the concept of dynamic slice in Fig. 2. This figure also shows how dynamic slices narrow down the fault space when compared to state-of-the-art static slice approach. For instance, during the time window C5, register *FF* (Line 27) is not in dynamic slice meaning that we do not need to inject fault in *FF* at C5 time window. Dynamic slice gives us critical faults and eliminates those faults that are not critical. In this way, we manage to reduce fault list by injecting only critical faults. This provides a speed-up in the fault injection simulation time as each injected fault increases total run time of fault injection campaign.

1) Implicit Fault Collapsing in Dynamic Slices: In our proposed methodology, dynamic slices cover both sequential and combinational parts. In this way, all faults outside of dynamic slices are 100% undetected and can be collapsed to exclude them from the fault list. When considering the average CPU time per a fault, an undetected fault spends more CPU time than a detected fault as the fault injection simulation for an undetected fault lasts until the end of the simulation. Hence, it is very effective to identify undetected faults without running fault injection campaigns.

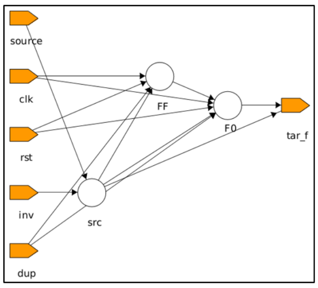


Fig. 3. Backward static slice on the signal *TAR_F* in the chopper design.

In the previous work [10], only sequential parts are considered in dynamic slices; however, both registers (sequential) and combinational parts that are connected to the registers are counted in dynamic slices in this work. In Fig. 4, dynamic slice is built by considering the register *inst_dest_bin* and *inst_dest* (combinational) so that we can have 100% accurate results. This is called as implicit fault collapsing since we avoid injections at time-steps as data inside registers is not being consumed.

D. Fault Injection Simulation

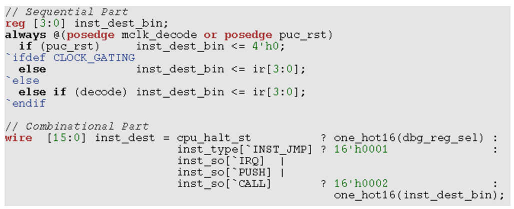
Fault injection enables to verify the capability of a safety mechanism to recognize failures in a design's functionality, by injecting faults into the design. In a fault injection simulation, target system and the possible hardware faults are modeled and simulated by the simulator. In this process, the system behaves as if there is a hardware fault.

To inject faults into a design, fault injection simulator needs to know fault target at which to inject fault. In this work, we enable fault instrumentation on the dynamic slices, more specifically on registers that are in dynamic slices. In other words, the proposed method identifies critical faults from dynamic slices and inject them at the specified times. As a fault model, we use Single Event Upset (SEU) fault type which inverts the value of output of a sequential element and hold the modified value until it is assigned a new value. Another thing that fault injection simulations need is an *observation point*, since the purpose of a fault campaign is to verify that an error will be observed at some specific point in the design. By defining explicit observation points when running a good simulation, we can generate data that will help us to determine if an injected fault is detected or undetected at one or more specified nets.

In brief, fault injection simulation is used to show the effectiveness of the proposed method. We inject one fault in one simulation run. Also, in the case of having more than one observation point in the analysis, the proposed method prevents multiple injection of faults within the overlap of static slices.

IV. Experimental Results

In order to verify the accuracy of the proposed fault injection method, we evaluate our application on industrial CPU with different workloads [24]. In the following subsections, firstly,



```
// Sequential Part
reg [3:0] inst_dest_bin;
always @(posedge mclk_decode or posedge puc_rst)
  if (puc_rst)      inst_dest_bin <= 4'h0;
`ifdef CLOCK_GATING
  else              inst_dest_bin <= ir[3:0];
`else
  else if (decode) inst_dest_bin <= ir[3:0];
`endif

// Combinational Part
wire  [15:0] inst_dest = cpu_halt_st          ? one_hot16(dbg_reg_sel) :
                         inst_type[`INST_JMP] ? 16'h0001               :
                         inst_so[`IRQ]  |
                         inst_so[`PUSH] |
                         inst_so[`CALL]       ? 16'h0002               :
                                                one_hot16(inst_dest_bin);
```

Fig. 4. Implicit fault collapsing.

we explain our experimental setup. Then, we show the results in detail.

A. Experimental Setup

Aiming to automate the execution of fault injection campaigns using the different tools, an application is developed as in the Fig. 5. This is the more detailed illustration of Fig. 1. We create generic scripts to activate the tools and automate the flow. In this work, the proposed methodology is integrated into Cadence flow but it can be applied using tools by any major EDA vendor.

In the first step, backward static slice is built for a selected observation point by using Cadence® JasperGold Formal Verification Platform. Then, we generate coverage results through Cadence® Xcelium™ for each clock cycle that defines the size of the dynamic slices. In the next step, static slice and executed statements data are sent to fault injection simulation to define fault target for the campaign. Annotation results provide information regarding to number of injected faults, number of detected and undetected faults. Moreover, we also use the profiling feature of the tool that measures where CPU time is spent during simulation. The profiler generates a run-time profile file that contains simulation run-time information that is useful for comparing execution time of different campaigns. Cadence® Xcelium Fault Simulator is used for fault injection simulations.

B. Evaluation and Results

We evaluate our methodology on a 16-bit microcontroller core [24] with a single address space for instructions and data. To show the effectiveness of the proposed method, we use three different workloads on *openMSP430*. We show our results in two categories as fault list reduction and time savings. Then, we evaluate the accuracy of this methodology by comparing our results to a state-of-the-art static slicing optimization method.

In the first step, backward static slice is built from *dmem_din* observation point which is the main output of the core and then coverage data is calculated. Next, considering the registers in static slice, instruction source and destination registers are selected as fault targets to apply the proposed method since

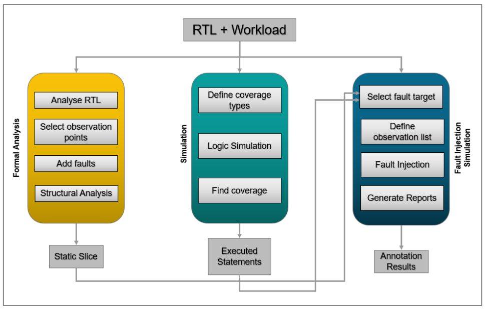


Fig. 5. Overall flow of experimental setup.

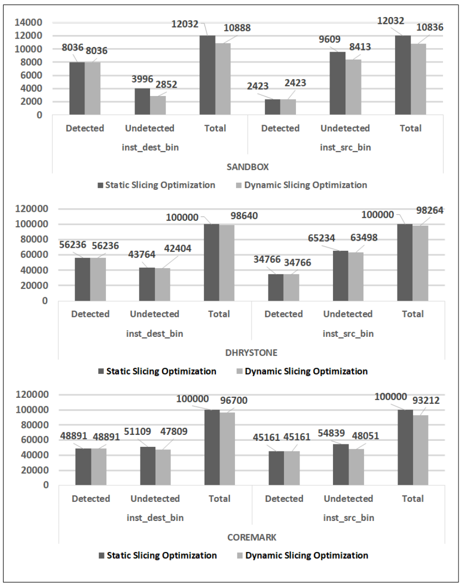


Fig. 6. Fault list reduction based on three different workloads.

these registers are widely used in fault injection applications as they hold all instructions.

Table I shows the comparison of two techniques: $a)$ state-of-the-art static slicing and $b)$ dynamic HDL slicing. We perform a fault injection campaign for each workload and a fault target (and) for each approach. For the execution of Dhrystone and Coremark workloads with static slicing, we select 100k faults after the warm-up phase of the CPU.

1) Fault List Reduction: Fig. 6 shows the reduction in the number of faults injected. All detected faults seen in Fig. 6 are critical faults. As seen in this charts, dynamic HDL slicing is effective in pruning the fault list as compared to the static slicing. Table II shows the percent reduction in the number of faults injection. The best reduction is achieved in Sandbox workload as a reduction of 9.94%. The magnitude of the fault list reduction depends on the workload characteristics. In this experimental results, the fault list reduction varies between 1.36% and 9.94%. These analysis reveal that dynamic HDL slicing prune the fault list and identify the critical faults successfully while analysis and optimization effort costs are very minor. Additionally, to identify undetected faults and exclude them from the fault list provides a increased fault coverage as it can easily be seen in Table I.

2) Time Savings: Table I shows the total CPU time of overall regression for each fault injection campaign. Dynamic slicing provides various time savings from 1.58% to 16.91% as shown in Table III. As in fault list reduction, time savings depend on the workload characteristic. When considering the need of multiple fault injection campaigns in real life applications, this time savings can expeditiously increase.

TABLE I
EXPERIMENTAL RESULTS ON OPENMSP430

	Sandbox		Dhrystone		Coremark	
	Static Slicing	Dynamic Slicing	Static Slicing	Dynamic Slicing	Static Slicing	Dynamic Slicing
inst_dest_bin						
Detected	8036	8036	56236	56236	48891	48891
Undetected	3996	2852	43764	42404	51109	47809
Total	12032	10888	100000	98640	100000	96700
Total CPU time of overall regression	1197.1s	994.7s	658919.7s	622459.0s	3437663.0s	3323109.9s
Fault Coverage	66.788%	73.806%	56.236%	62.735%	48.891%	50.559%
inst_src_bin						
Detected	2423	2423	34766	34766	45161	45161
Undetected	9609	8413	65234	63498	54839	48051
Total	12032	10836	100000	98264	100000	93212
Total CPU time of overall regression	1488.2s	1284.2s	803009.1s	790300.0s	3575198.1s	3178378.2s
Fault Coverage	20.137%	22.361%	34.766%	35.380%	45.161%	48.450%

TABLE II
PERCENTAGE OF REDUCTION OF THE TOTAL NUMBER OF INJECTIONS WITH DYNAMIC HDL SLICING

	Sandbox	Dhrystone	Coremark
	Percentage of reduction	Percentage of reduction	Percentage of reduction
inst_dest_bin	9.51%	1.36%	3.3%
inst_src_bin	9.94%	1.74%	6.79%

TABLE III
TIME SAVINGS USING DYNAMIC HDL SLICING

	Sandbox	Dhrystone	Coremark
	Dynamic slicing time saving	Dynamic slicing time saving	Dynamic slicing time saving
inst_dest_bin	16.91%	5.53%	3.33%
inst_src_bin	13.71%	1.58%	11.10%

3) Accuracy: In this work, we show the results of a fault injection campaign performed using dynamic slicing, along with a state-of-the-art static slicing approach. These results reveal that dynamic slicing achieves the same number of detected faults as static slicing campaign. This means that dynamic slicing can be used for different purposes as it is an accurate fault injection methodology. For instance, SFI [8] prunes the fault list in terms of margin of error with a given confidence level. However, dynamic slicing exclude only non-critical faults and find all critical faults with a 100% accuracy.

V. CONCLUSIONS

Fault injection on RTL requires excessively long simulation time which prevents detailed reliability evaluation of hardware components with significant number of injections. This paper presents a method to speed-up fault injection campaigns by minimizing of fault injection locations. The method applies dynamic slicing on HDL to accurately pinpoint fault injection locations and allows injection of critical faults in these time windows. In this way, this paper narrows down the fault space and provides reduced simulation time. Moreover, average 5-10% extra gain in simulation time for fault injection is a significant improvement of the total chip validation costs, as this phase is the most time consuming. The proposed method is language-agnostic and suitable for industrial grade EDA tool flows. Experimental results on industrial-size example show that we achieve significant speed-up of the fault injection simulation when comparing to the state-of-the-art flows.

ACKNOWLEDGEMENT

This research was supported by project RESCUE funded from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

REFERENCES

[1] ISO, "ISO 26262 - road vehicles - functional safety," Dec 2018.
[2] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: a flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 248–260, Feb 1995.
[3] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "An fpga-based approach for speeding-up fault injection campaigns on safety-critical circuits," *Journal of Electronic Testing*, vol. 18, no. 3, pp. 261–271, Jun 2002. [Online]. Available: https://doi.org/10.1023/A:1015079004512
[4] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin, "Crashtest: A fast high-fidelity fpga-based resiliency analysis framework," in *2008 IEEE International Conference on Computer Design*, Oct 2008, pp. 363–370.
[5] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *11th IEEE International On-Line Testing Symposium*, July 2005, pp. 260–265.
[6] G. Fey and R. Drechsler, "A basis for formal robustness checking," in *9th International Symposium on Quality Electronic Design (isqed 2008)*, March 2008, pp. 784–789.
[7] L. Berrojo, I. Gonzalez, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, March 2002, pp. 847–852.
[8] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.
[9] J. Raik, U. Repinski, M. Jenihhin, and A. Chepurov, "High-level decision diagram simulation for diagnosis and soft-error analysis," *Design and Test Technology for Dependable Systems-on-Chip*, pp. 294–309, 2011.
[10] A. C. Bagbaba, M. Jenihhin, J. Raik, and C. Sauer, "Efficient fault injection based on dynamic hdl slicing technique," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, July 2019, pp. 52–53.

[11] V. Sieh, O. Tschache, and F. Balbach, “Verify: evaluation of reliability using vhdl-models with embedded fault descriptions,” in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, June 1997, pp. 32–36.

[12] P. Folkesson, S. Svensson, and J. Karlsson, “A comparison of simulation based and scan chain implemented fault injection,” in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, June 1998, pp. 284–293.

[13] D. T. Smith, B. W. Johnson, J. A. Profeta, and D. G. Bozzolo, “A fault-list generation algorithm for the evaluation of system coverage,” in *Annual Reliability and Maintainability Symposium 1995 Proceedings*, Jan 1995, pp. 425–432.

[14] X. Iturbe, B. Venu, and E. Ozer, “Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu,” in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Sept 2016, pp. 91–96.

[15] R. Travessini, P. R. C. Villa, F. L. Vargas, and E. A. Bezerra, “Processor core profiling for seu effect analysis,” in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, March 2018, pp. 1–6.

[16] A. Evans, M. Nicolaidis, S. 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)*, March 2013, pp. 727–732.

[17] W. Mansour, R. Velazco, R. Ayoubi, H. Ziade, and W. E. Falou, “A method and an automated tool to perform set fault-injection on hdl-based designs,” in *2013 25th International Conference on Microelectronics (ICM)*, Dec 2013, pp. 1–4.

[18] T. Bonnoit, A. Coelho, N. Zergainoh, and R. Velazco, “Seu impact in processor’s control-unit: Preliminary results obtained for leon3 soft-core,” in *2017 18th IEEE Latin American Test Symposium (LATS)*, March 2017, pp. 1–4.

[19] M. Jenihhin, A. Tšepurov, V. Tihhomirov, J. Raik, H. Hantson, R. Ubar, G. Bartsch, J. H. M. Escobar, and H. Wuttke, “Automated design error localization in rtl designs,” *IEEE Design Test*, vol. 31, no. 1, pp. 83–92, Feb 2014.

[20] U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. D. Guglielmo, G. Pravadelli, and F. Fummi, “Combining dynamic slicing and mutation operators for esl correction,” in *2012 17th IEEE European Test Symposium (ETS)*, May 2012, pp. 1–6.

[21] E. M. Clarke, M. Fujita, S. P. Rajan, T. W. Reps, S. Shankar, and T. Teitelbaum, “Program slicing for vhdl,” *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 125–137, 2002.

[22] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura, “Program slicing on vhdl descriptions and its applications,” 1996.

[23] B. Korel and J. Laski, “Dynamic program slicing,” *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(88)90054-3

[24] (2019) Opencores. [Online]. Available: http://www.opencores.org, last accessed July 24, 2019

Appendix 7

VII

Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Annachiara Ruospo, Riccardo Mariani, Ghani Kanawati, Ernesto Sanchez, Matteo Sonza Reorda, Maksim Jenihhin, Said Hamdioui, and Christian Sauer. Special session: Autosoc - a suite of open-source automotive soc benchmarks. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–9, 2020

2020 IEEE 38th VLSI Test Symposium (VTS)

Special Session: AutoSoC - A Suite of Open-Source Automotive SoC Benchmarks

Felipe Augusto da Silva*†, Ahmet Cagri Bagbaba*‖, Annachiara Ruospo‡, Riccardo Mariani¶, Ghani Kanawati§, Ernesto Sanchez‡, Matteo Sonza Reorda‡, Maksim Jenihhin‖, Said Hamdioui† and Christian Sauer*

*Cadence Design Systems, Munich, Germany
†Delft University of Technology, Delft, The Netherlands
‡Politecnico di Torino, Turin, Italy
§ARM Ltd, Austin, TX, USA
¶Nvidia Corporation, Milan, Italy
‖Tallinn University of Technology, Tallinn, Estonia

***Abstract*—The current demands for autonomous driving generated momentum for an increase in research in the different technologies required for these applications. Nonetheless, the limited access to representative designs and industrial methodologies poses a challenge to the research community. Considering this scenario, there is a high demand for an open-source solution that could support development of research targeting automotive applications. This paper presents the current status of 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. This paper describes the available configurations of the benchmark including an initial assessment for ASIL B to D configurations.**

Keywords - Automotive benchmark; SoC; open-source; Functional Safety; ISO 26262.

I. INTRODUCTION

In recent years, advances in technology enabled the employment of automated systems to control driving tasks. The idea of electronic devices having full control over a vehicle promises to change the concept of mobility in the near future. However, allowing computers to control all the tasks in a vehicle requires high complexity systems and major concerns with respect to the safety. The development of Autonomous Vehicles applications, where a system failure could cause life-threatening situations, entails in state-of-the-art challenges on different aspects of system development. Concerns with Reliability, Security, Quality, and compliance to Safety Standards are of high priority. This scenario requires adoption of new techniques and methodologies that will facilitate development and verification of these applications. Several organizations are working to close the technological gap for Autonomous Vehicles. However, in order to assess the quality of the proposed solutions, it is necessary to compare the results against what is applied in the industry. Nowadays, development lifecycles and verification techniques applied by industry are not disclosed, and each big player in the automotive sector has its own methodologies and tools. In addition, there is limited access to automotive hardware and software solutions. This is a challenge for researchers, that may not be able to 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. It should be outlined that the benchmarks should include not only the hardware description (at different levels of abstraction), but also compatible software modules (Operating System, peripheral drivers, sample applications) and information about the implemented safety and security mechanisms.

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 [1], [2], software design [3], operational systems [4], among others. [5] provides a broader look on specific reliability challenges for autonomous systems, for both automotive and robotics. The challenges of Functional Safety compliance, based on standards like ISO 26262, are also explored in research as [6], [7]. The authors point out Fault Injection (FI) Simulation as one of the critical steps for compliance with the standard. For that reason, different approaches are proposed to leverage FI Simulation, optimization of the simulation techniques [8], [9], combination of multiple fault analysis technologies [10], analysis of faults on different hardware abstractions levels [11], [12], and many others. Several works are also discussing the security issues imposed by these applications [13]. Challenges with hardware attacks [14] and secure in-vehicle communication [15] are being investigated and their interference with functional safety and reliability is getting to the front. 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 Systems on Chip (SoCs) with an operating system and software applications that are representative of the Automotive sector. 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.

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 the analysis of commercial solutions, and considering common development techniques deployed by 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, RT and gate level). The suite includes multiple configurations with different levels of Safety Mechanisms (SMs), enabling investigation of Functional Safety aspects. The AutoSoC appears as an interesting 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*.

The remainder of this paper is organized as follows. Section II elaborates on the reasons behind the need for standardization and benchmarking in the automotive as well as in the closely related robotics domain. Next, Section III describes the definition of the functional requirements for the AutoSoC based on the characterization of industrial solutions. Afterwards, in Sections IV, V and VI, we describe its base HW and SW components, the Safety components and the available benchmark configurations. Section VII outlines a preliminary functional safety analysis targeting different ASIL configurations. Last, Section VIII presents our conclusions and future work.

II. SAFETY STANDARDIZATION AND BENCHMARKING FOR AUTOMOTIVE AND ROBOTICS

Nowadays, highly automated safety-critical systems (such as autonomous vehicles and autonomous mobile robots) are implemented with very complex integrated circuits. They are composed of a large set of HW elements, executing an equally large set of SW elements, often from third parties. This complexity has created a strong demand for standardization initiatives related to semiconductors, to guarantee uniformity, interoperability and repeatability of the many activities required by a safety lifecycle. The main initiative is the 2nd edition of ISO 26262, with a part 11 [16] fully dedicated to the application of ISO 26262 to semiconductor technologies. The part 11, with its 179 pages, provides a detailed set of guidelines on principles, methods and architectures for digital, mixed signal, programmable device and sensor type of integrated technologies. The variety of solutions and combinations provided by part 11 is huge, as also the opportunity to create new ideas fulfilling the principles highlighted by the standard.

On the other hand, that vastity of options is a challenge from several points of view. For example, despite the ISO 26262 provides a mathematical approach to quantify the probability of failure due to HW random failures, it is very effort intensive to apply it and quickly compare the effectiveness of each proposed solution. In fact, the results are highly dependent on the chip architecture and the related SW application executed on it. The same challenge exists for the verification activities (e.g. fault injection) required to confirm the effectiveness of some of the functional safety properties, such as the diagnostic coverage. The time spent to setup each fault injection campaign for each different architecture solution makes unpractical to use it during the exploration phase – so limiting the creativity and the space of possible solutions. 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 how the SW redundancy is executed (e.g. if it is a task per task or instruction per instruction redundancy, if the OS is in common or shared, etc.), on how often the two SW executions are compared, on how many variables of the compared SW are exposed to the comparison, etc. It is also 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 Safety Mechanisms to compare and detect the potential failure. As also it is necessary to evaluate the degradation of performance (e.g. in terms of worst case execution time or WCET) that the comparisons of the loosely coupled lock-step are causing to the data traffic of the nominal functionality.

The complexity described by the previous examples indicates the strong need of an open-source benchmarking environment, to provide scientists with a ready-to-use and clearly defined platform on which 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 SW test libraries (STLs), a well-known method described in ISO 26262 part 11. The availability of a common benchmark will allow a transparent and well defined comparison of the impact to the application caused by two different STL implementations.

III. 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.

A. *Industry Solutions Characterization*

Nowadays, 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, automatization of driving tasks, among others. All those features require robust solutions that must consider aspects of functional safety and security. Although different commercial solutions are available, in general, architectures have similarities that can be explored to define a set of requirements for an Automotive SoC. The requirements for the AutoSoC were gathered based on an analysis of the datasheets of commercial Automotive SoCs. We considered the main characteristics of available solutions to identify common aspects that can be regarded as mandatory by the industry. In general, the analysis can be split into 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 [17] includes two CPUs for common applications and an additional Dual Lockstep CPU for safety-critical applications. The Infineon AURIX [18] and Texas Instruments TDA2SG [19], follow a similar concept by including a CPU and separated cores for dedicated functionalities. Dual-Core Lockstep (DCLS) is the most common safety mechanism available for CPUs. For the memories, including RAMs and caches, industrial solutions usually deploy Error Correction Codes (ECCs) and Parity. DCLS, ECCs, and Parity have an advantage regarding Functional Safety analysis. These SMs are introduced by the recommendations of ISO 26262 [20] and include a reference of their fault coverage capabilities. Hence, by deploying any of these SMs as described in ISO 26262, the referenced Diagnostic Coverage can be directly used during Functional Safety Analysis.

The other components available in the analyzed SoCs could be categorized as communication protocols, application-specific, security, and infrastructure peripherals. In general, the commercial solutions implement a good variety of communication peripherals, including automotive protocols as CAN and FlexRay, and general protocols as Ethernet, SPI, and I2C. Another common characteristic is the availability of Video and Audio dedicated hardware. As the majority of the SoCs aim to Advanced Driver-Assistance Systems (ADAS) applications, they include peripherals like GPUs, video codecs, Image Processing Units, and Audio DSPs. In the security domain, apart from proprietary features that are not detailed, the most common components are cryptography engines, like Advanced Encryption Standard (AES), Data Encryption Standard (DES), Hash, among others. 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.

TABLE I
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	+	-	+

Considering the characteristics of the evaluated commercial solutions, it is possible to define a common set of features that can be seen as required by the automotive industry. The addition 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 found in the evaluated commercial solutions is available in Table I.

B. *AutoSoC Functional Blocks*

Based on the characterization of industrial solutions, summarized in Table I, an initial architecture of AutoSoC was established. Functional blocks were defined aiming to cover the minimum set of features required for a representative automotive benchmark suite. The concept of functional blocks is also important to keep the design modular. Different versions of AutoSoC can deploy diverse hardware components to cover the requirements of each functional block. Figure 1 illustrates the outcome of our analysis.

As it happens in most commercial solutions, the AutoSoC has two main processing units. The *Safety Island* is responsible for all safety-critical processing capabilities. It is composed of CPUs and memories that must be covered by Safety Mechanisms according to the requirements of ISO 26262. The division between safety-related hardware and the rest of the SoC components supports the compliance with Functional Safety standards, as only the safety-related hardware is required to comply with ISO 26262. The other processing unit is the *Application Specific Block*. This unit implements the

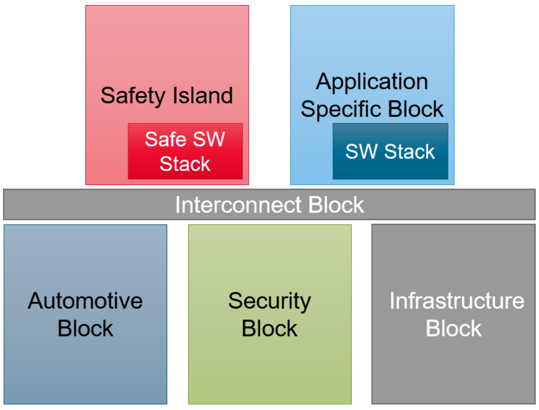


Fig. 1. AutoSoC Functional Blocks.

hardware required for application-specific processing. It may include CPUs and memories for high demand applications, GPUs and Image processing units for video applications, among others. The target functionality for each given AutoSoC configuration will define the Hardware components required for the *Application Specific Block*. Also, it is important to notice that the *Safety Island* and the *Application Specific Blocks* have dedicated Software stacks. Both can execute distinct 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 to perform 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 aiming at support the new security standard under development ISO 21434. The latter aims at defining a Cybersecurity Assurance Level (CAL), similar to the ASIL concept [21]. The *Infrastructure Block* is responsible for the on-line health monitoring of the SoC. It includes debugging features such as JTAG and UARTs to ease the development process. Finally, the *Interconnect Block* is responsible for internal SoC communication. It may deploy common communication buses, like AXI and Wishbone, or more advanced options such as a Network-on-Chip (NoC).

IV. 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 configurations to be instantiated by simply enabling additional Safety components. The next sections describe the available Safety components and AutoSoC configurations.

A. 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, RT level, and gate level). A further requirement is that the CPU has to be open-source. Different analyzed options could be considered as good candidates for the CPU. For instance, the Amber2 [22] is a 32-bit RISC CPU compatible with the ARM v2a instructions set. Another considered option was the Gaisler LEON3 [23]. It includes a 7 stages pipeline, a comprehensive set of peripherals, and support scripts. This work has deployed the OpenRISC [24] (mor1kx implementation) as the main CPU. The OpenRISC includes a better variety of support tools, an active community and the resources for the development of a Virtual Platform. Also, the community supports a variety of compatible peripherals that can be easily integrated, including CAN, AES, and DES [25].

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, all 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 was used as a base for the AutoSoC. By deploying the example, we can cover the infrastructure and interconnect blocks. Also, we can reuse part of the provided test environment to speed up the development.

B. 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 fault. In the case of an SoC, the software application executed by the CPU defines the functionality. For that reason, the software stack is an important 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. The Baremetal folder includes development resources as Makefiles, drivers, and around 50 compiled test applications. Also, a compiled Linux kernel (bootable in simulation) is available in the Linux folder.

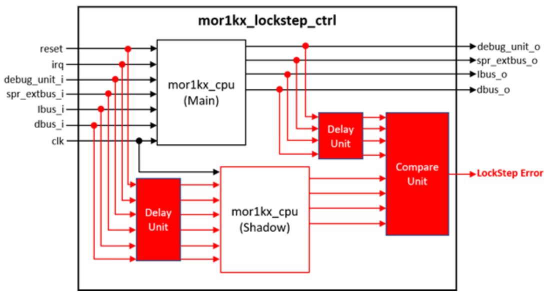


Fig. 2. Time diversity Dual-Core Lockstep implementation.

Furthermore, the RTEMS folder includes a development environment with Makefiles, drivers, and applications. Finally, an *Automotive Cruise Control* application was developed. The application is based on the RTEMS Operational System. It comprises four real-time tasks for reading vehicle sensor data, computing actuation, setting some engine parameters, and housekeeping.

V. AUTOSOC SAFETY COMPONENTS

Another important aspect of the benchmark is the availability of Safety Mechanisms in the Safety Island. As this block is responsible for executing safety-critical applications, we need to assure 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. Different safety mechanisms schemes were conceived, each targeting different Automotive Safety Integrity Levels (ASIL).

A. *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. The implementation of the DCLS with time diversity is illustrated in Figure 2.

The performance of the main 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 outputs 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 the their outputs [26]. To minimize the potential of common-mode failures the DCLS mechanism includes time diversity. Time diversity works by applying a delay in the execution of the shadow processor. The delay is obtained by including a delay unit in the driven signals of the CPU. Delay units are also added to the outputs of the main processor, to align both core outputs for the Compare Unit. The Delay Units can be configured with the desired time shift: 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 input of the Compare Unit, a mismatch between the CPU outputs 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.

B. *Software Test Libraries*

A Software Test Library, also referred to as STL, is a collection of software tests that are run on power-on (key-on), power-off (key-off) or periodically to prevent faults from leading to single-point failures or prevent them from becoming latent as a result of a multiple-point fault.

This software mechanism aims at detecting permanent faults that can occur anytime during the execution of a safety application and can cause a safety violation. An STL corresponds to a set of software procedures, usually developed in assembly code, C code or a combination of both. These may be executed either at boot-time or run-time. In the former case they require supervisor capabilities and therefore, to avoid conflict with the Operating System (OS), are usually executed during the power-on and power-off. On the other hand, when the STLs are executed at run-time, they have to coexist with the OS. Then, it is essential to make these tests run in a short period of time, usually few milliseconds, to avoid affecting the behavior of the other software applications running on the same hardware. The software scheduler will schedule these tests at specified time intervals when the hardware is idle or running less time sensitive applications.

In the recent years, several semiconductor and IP companies started to provide their customers with Software Test Libraries (STLs) to be used for on-line fault detection when the target devices are used in safety-critical applications. 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 exactly 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 at the moment and requires special skills in order to achieve sufficiently high fault coverage figures. Computing these figures for a given STL also requires a new generation of tools called Functional Fault Simulators. Several recent works introduced guidelines on how to correctly generate STLs for CPUs [27], [28] and peripherals [29], how to speed up the FI experiments [30], how to maximize their fault coverage in the different scenarios (possibly minimizing the test time [31]), and how to re-use existing STLs.

C. 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, the internal memories are a primary target for SMs. The ISO 26262 standard includes recommendations for well-known memory Safety Mechanisms. Based on the recommendations and the findings of the industry solutions characterization, Error-Detection-Correction Codes (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 RT level representation of the CPU. The deployment of an SM with high Diagnostic Coverage, like ECC, on all internal memories, will provide a satisfying coverage for the overall CPU.

D. External Memory ECC

The other elements of the Safety Island must also be verified 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 the memory for storing data and control parameters. As the software application function relays on the external RAM, memory failures have a direct impact on the intended functionality. The external RAM must also be covered by ECC to avoid propagation of internal memory faults to the outputs of the Safety Island.

E. 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 to the memory and would not the detected by their SM. To avoid these cases, a parity checker was included to cover data transmissions between CPU and memory. The Parity checker monitors data bus transmissions, and calculates a Parity bit for all communications between CPU and memory. The Parity bit is transmitted by a direct connection between the Parity Check blocks. In case of a wrong parity, an alarm is set to inform the system.

F. Checkpoint Control

Even if the DCLS SM is employed, both CPUs could get stuck in the same software instruction, and none of the mentioned SMs would be able to detect this fault. For that reason, a Checkpoint Control safety mechanism was implemented. The Checkpoint control monitors the Data Bus expecting predetermined software signatures in specific memory locations. 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 is capable of verifying not only if the software application is running, but also if the Control Flow is as expected. The Checkpoint Control is fully customizable during elaboration, allowing the definition of the software signatures, expected sequence, and deadlines.

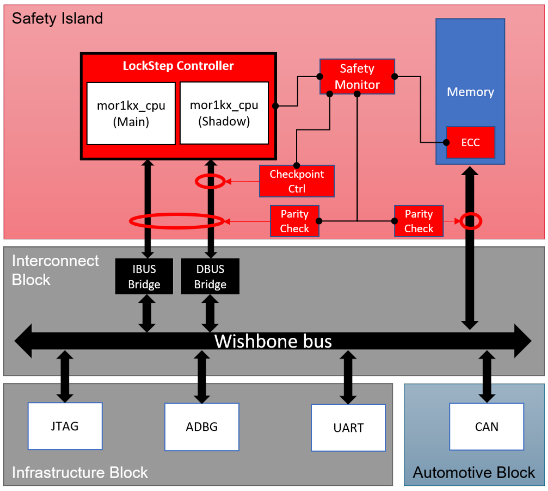


Fig. 3. AutoSoC Safe Configuration.

G. Safety Monitor

Finally, a Safety Monitor block was developed 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 illustrates the architecture of the AutoSoC Safe configuration, including the DCLS, External Memory ECC, Bus Parity and Checkpoint Control.

VI. AutoSoC Configurations

This section outlines the available benchmark configurations 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 1, will be developed in the next stages of our work. The modular design of the AutoSoC allows the reuse of the Functional Safety Analysis, performed in the scope of this paper, 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, choose the software application and enable any combination of Safety Mechanisms by adding defines to the 'plus args' config file

TABLE II
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	+	-	-	+	+	+

(e.g. +define+DCLS). The new configuration can then be elaborated and simulated with the provided Makefile. Although any possible combination of components can be created, we have defined a group of initial configurations for the AutoSoC. These configurations are based on common SM combinations from industry solutions. Table II illustrates some potential configurations for the AutoSoC. For the scope of this paper, we have performed a preliminary safety assessment for three configurations. The configurations AutoSoC ECC, AutoSoC STL, and AutoSoC DCLS, were analyzed as candidates to target different ASIL levels.

VII. PRELIMINARY FUNCTIONAL SAFETY ANALYSIS

This section describes the functional safety analysis of some of the available configurations of AutoSoC. Functional Safety Analysis, as specified by ISO 26262, 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 Automotive Safety Integrity Level (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 Safety Mechanisms (SMs). The required percentage of detection, or Diagnostic Coverage (DC), is defined by the ASIL.

Typically, Functional Safety analysis is completed at later stages of the hardware design. Additional parameters like area, Failure-in-Time (FIT) rate, and Failure Modes distribution, are necessary to confirm design compliance to the required ASIL. These parameters are used 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 next step of our work is to finalize the gate-level description of AutoSoC, determine the possible failure modes, define the diagnostic coverage based on the failure mode distribution, and calculate the final safety metrics.

A. *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 is high, meaning 99% of detection for random hardware faults. 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.

TABLE III
DCLS CPU FAULT COVERAGE

Fault Target	SA(1/0) Faults	Detected by DCLS	Residual Faults
mor1kx_cpu	675,504	668,749	6,755

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 99% of the faults in the mor1kx_cpu(Main) will be detected by the Lockstep Controller. With 99% of fault coverage, we can expect the AutoSoC DCLS to be a good candidate to comply with ASIL D requirements. Table III illustrates the potential fault coverage for the AutoSoC DCLS configuration.

B. *AutoSoC ECC configuration*

As described for the Processing Units, ISO 26262 also includes recommendations of Safety Mechanisms 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 every one and two-bit failures, and some three or more bit failures in a word. The standard defines the typical diagnostic coverage for ECC is also 99% of detection for random hardware faults. Usually, on complex CPUs, internal memories, or caches, occupy the largest area on the physical devices. For that reason, they will have a high contribution to the design Failure-In-Time (FIT) rate. This contribution will appear in the Failure Modes (FM) distribution, with cache-related FMs requiring Safety Mechanisms to decrease the residual FIT. It is a common design practice to protect the cache memories with ECC or Parity. In the AutoSoC design, the internal memories represents a potential of 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

TABLE IV
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**

SM to the internal memories represent a good overall coverage for the CPU faults. The AutoSoC internal ECC configuration considers the incorporation of ECCs to all internal memories. Table IV demonstrates the fault coverage of the ECC for each internal 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 a good candidate to comply with ASIL B requirements.

C. *AutoSoC STL configuration*

To avoid the hurdle of the extra hardware required by DCLS schemes, there is an increasing demand for software strategies for the on-line testing of automotive processors. This section describes the main characteristics of the software test libraries being developed to improve the AutoSoC CPU fault coverage and reports the preliminary results.

Preliminary results are gathered 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 [32]: ATPG-based, deterministic and evolutionary-based [33]. 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. Indeed, when *evaluating* the test programs fault coverage, the exact stream of instructions entering the pipeline must be deterministic: these modules might lead to a fluctuating fault coverage and therefore should be deactivated for the fault grading process (which directly contributes to the ASIL process certification) [34]. This does not prevent the caches (or similar) from being used when the STL is integrated in the application software and deployed in field.

Starting from these considerations, permanent faults injection analyses have been carried out on a total of 42,160 faults target for the mor1kx_cpu at RT level, and a total of 60,672 permanent faults for the mor1kx_alu and the mor1kx_lsu units at gate level. If considering the mor1kx_alu and mor1kx_lsu at RT level, there are 4,938 fault targets. The Fault Injections experiments were performed at both the RT and gate level, mimicking the typical process used in practice, where RT level estimations are used as a proxy for gate level fault

TABLE V
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

coverage estimation during the STL development process. A further investigation was performed in order to identify all the untestable and safe faults [35], revealing a non-negligible increase in the fault coverage of the two targeted modules. Once again, the identification of untestable and safe faults represents a common issue in practice, given that their number may often be non negligible. Table V sums up the gathered results showing the achieved fault coverage on the ALU and LSU modules, both at the RT and gate level. The achieved Fault Coverage (FC) considering the redundant and safe faults is reported as Testable Fault Coverage (TFC).

The deployment of software routines to identify permanent faults is shown to be effective in multiple units of a CPU [35]. 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 Safety Mechanisms. A common practice in the automotive industry is to combine STLs with ECC in the internal memories of the CPU. For instance, in [35] 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 stuck-at-0 and stuck-at-1 faults, not considering the internal memories. If we consider the fault coverage from [35], the STL would be able to detect 35,583 faults. If we include the STL routines in the AutoSoC ECC ConfigurationVII-B, the combined Safety Mechanisms 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.

VIII. 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 the concerns with 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 that is based on commercial solutions, includes all essential components, is highly customizable, and allows comparability between distinct methodologies and results. This paper 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. Further works on AutoSoC may focus on new Safety Mechanisms or combinations of them, new techniques to automate the safety analysis (e.g., to better identify untestable and safe faults) and make it faster (e.g., speeding up functional fault simulation), and to evaluate cross-layer solutions to evaluate and increase the system dependability. We believe that the availability of this benchmark suite will allow researchers to develop new solutions and to quantitatively assess their effectiveness, thus contributing to the advancement of the state of the art in the area.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 722325.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[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.

[10] 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.

[11] 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.

[12] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," in *2016 Forum on Specification and Design Languages (FDL)*. IEEE, sep 2016.

[13] M. Singh and S. Kim, "Security analysis of intelligent vehicles: Challenges and scope," in *2017 International SoC Design Conference (ISOCC)*. IEEE, nov 2017.

[14] 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.

[15] 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.

[16] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 11: Guidelines on application of ISO 26262 to semiconductors*, International Standardization Organization Std., Dec. 2018.

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

[18] 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/

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

[20] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 5: Product development at the hardware level*, International Standardization Organization Std., Dec. 2018.

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

[22] *Amber 2 Core Specification*, opencores.org, Mar. 2015.

[23] Cobham Gaisler, *LEON3 Multiprocessing CPU Core*, 2010.

[24] D. Lampret et al., *OpenRISC 1000 Architecture Manual*, revision 0 ed., opencores.org, Dec. 2012.

[25] OpenRISC, "OpenRISC Community," 2020. [Online]. Available: https://github.com/openrisc

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] A. Floridia *et al.*, "Deterministic cache-based execution of on-line self-test routines in multi-core automotive system-on-chips," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020. Accepted for publication, to appear.

[35] 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.

Appendix 8

VIII

Felipe Augusto da Silva, Ahmet Cagri Bagbaba, Sandro Sartoni, Riccardo Cantoro, Matteo Sonza Reorda, Said Hamdioui, and Christian Sauer. Determined-safe faults identification: A step towards ISO26262 hardware compliant designs. In *2020 IEEE European Test Symposium (ETS)*, pages 1–6, 2020

Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs

Felipe Augusto da Silva*†, Ahmet Cagri Bagbaba*, Sandro Sartoni‡, Riccardo Cantoro‡, Matteo Sonza Reorda‡, Said Hamdioui† and Christian Sauer*

*Cadence Design Systems, Munich, Germany
†Delft University of Technology, Delft, The Netherlands
‡Politecnico di Torino, Turin, Italy

***Abstract*—The development of Integrated Circuits for the Automotive sector imposes on major challenges. ISO26262 compliance, as part of this process, entails complex analysis for the evaluation of potential random hardware faults. This paper proposes a systematic approach to identify faults that do not disrupt safety-critical functionalities and consequently can be considered Safe. By deploying code coverage and Formal verification techniques, our methodology enables the classification of faults that are unclassified by other technologies, improving ISO26262 compliance. Our results, in combination with Fault Simulation, achieved a Diagnostic Coverage of 93% in a CAN Controller. These figures allow an initial assessment for an ASIL B configuration of the IP.**

Keywords - ISO26262; Safe Faults; Fault Injection; Formal Methods; Simulation; Functional Safety; Verification.

I. INTRODUCTION

The increasing complexity in automotive applications is causing a shift in the traditional design flow. 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 be able to detect an extremely high percentage of potential faults while already deployed in the field. In the most advanced automotive ICs, where millions of design components are susceptible to random hardware faults, this process becomes challenging. Also, the demands for fault detection during the operational life of the design requires the deployment of suitable test mechanisms, as Self Test Libraries (STL). In operational mode, Design for Testability (DfT) often is not an option, as it could disturb the intended functionalities. Today, the approach based on STLs is widely adopted in the automotive industry [1][2][3].

Usually, Fault Injection (FI) Simulations are deployed for evaluation of the fault effects in the operational mode. However, FI Simulation alone is not enough to fully classify all faults. For those which are not detected we must rely on alternative analysis methods that can prove whether they could disturb safety-critical functionalities or not (*Safe Faults*). Previous works [4] showed that the number of Safe Faults can be significant in real applications. In complicated designs, manual analysis of fault effects is an arduous task that requires extensive knowledge of the design functionalities. Therefore, there is a high demand for a systematic approach for the identification of Safe Faults, allowing the reduction of manual efforts and improving compliance with Functional Safety standards.

Fault Injection (FI) Simulation is a state-of-the-art method for Functional Safety Verification, being recommended by ISO26262. As such, several researchers explored the optimization of FI campaigns [5][6][7][8]. The main purpose is to show that fault effects are observable on safety-related outputs of the design. In case an injected fault is not observable, it must be re-analyzed. Nonetheless, observation or detection of all design faults is usually not possible. Therefore, alternate methods are necessary for the classification of residual faults. Formal Methods can be employed to leverage the classification of faults. The ability of formal techniques in analyzing the design behavior for all possible combinations of inputs can help to identify Safe Faults [9][10][11]. These faults cannot be tested by ANY valid test stimuli. Faults that are untestable can also be described as Structural-Safe Faults. The combination of FI Simulation and Formal techniques was also examined [12][13][14][15]. The mixed technologies approach is usually deployed to improve the classification of faults. However, even with the identification of Detected and Structural-Safe Faults, there are still residual faults that require further classification. To avoid manual analysis of fault effects and still fulfill ISO26262 requirements, a different methodology is needed.

Our work tackles the classification of residual faults. We propose a methodology that identifies design elements where a fault cannot disturb the safety-critical outputs of the design. In case the effect of a fault does not affect safety-related functionalities, there is no chance of Safety Goal violations. Therefore, these faults can be classified as Determined-Safe. Different from Structural-Safe Faults which cannot be tested by any functional test stimuli, Determined-Safe Faults may affect the output of the design. However, they cannot affect safety-critical functionalities. Initially, we deploy code coverage techniques to identify design elements that are not exercised during functional verification. The candidates are examined by code inspection and simulation. If confirmed that the candidates are not safety-related, they are translated into formal rules. Finally, we configure all the rules in a Formal analysis tool for the identification of Determined-Safe Faults. The main contributions of this work are:

- A systematic approach for classification of Faults that cannot affect safety-critical functionalities;

- Demonstration of the proposed methodology using an automotive CAN Controller IP;
- Improving the fault classification to 93% of Diagnostic Coverage, achieving ASIL B requirements out of the box.

The rest of the paper is organized as follows: Formal techniques for identification of Safe Faults are introduced in Section II. Section III describes the proposed methodology. Section IV explains the validation process and discusses our results. Section V concludes.

II. Fault Classification by Formal Tools

Fault classification is a strenuous task. A fault can only be labeled as Safe if one can prove that it cannot be tested by ANY functional test stimuli. The formal analysis appears as a good alternative for this purpose since it is not limited to a specific time or state. Instead, the scope is global, and every evaluation context and test stimuli is considered [9]. Consequently, formal analysis can exhaustively prove that a fault can never produce any failure. This class of untestable faults can be classified as Structural-Safe.

Different EDA vendors explore fault analysis capabilities in their formal solutions. Generally speaking, these solutions automatically generate properties, not requiring knowledge of formal languages. 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.

A. *Structural Analysis*

The Structural Analysis aims to determine the testability of faults. The testability of the faults is determined by verifying the physical characteristics of the design. Figure 1 illustrates the examination applied by the Structural Analysis.

Figure 1 represents a circuit with combinational logic (g), inputs (in), outputs (out) and fault targets (f). Considering this circuit, it is possible to define the following fault behaviors by applying Structural Analysis:

1) As the only Observation Point (strobe) configured for the fault analysis is 'out0', any fault that is outside of its Cone of Influence is considered Untestable. For that reason, any fault in 'f1' is Structural-Safe as there is no physical connection between the fault location and the strobe.
2) Depending on the characteristics of 'g1' drivers, it is possible to define the activatability of 'f2'. For example, if 'g1' always output the logic value one, 'f2' would not be activatable for Stuck-at-1 faults. Consequently, a Stuck-at-1 fault in 'f2' would be Structural-Safe.
3) Characteristics of the combinational logic 'g2' could block 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'. Therefore, 'f3' would be Structural-Safe for Stuck-at-1 and Stuck-at-0 faults.

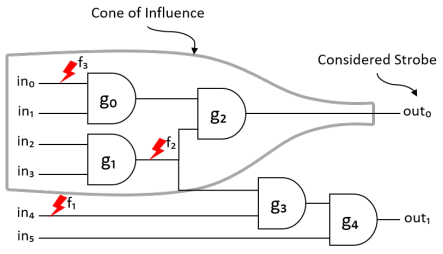


Fig. 1. Structural Analysis Example.

B. *Formal Analysis*

The Formal Analysis deploys formal techniques to investigate the behavior of a design under fault. The fundamental theory consists in creating a representation of the boolean function implemented by the design under test, where formal proves can be deployed. Modern Formal tools employ different formal techniques to achieve better performance. Although details of implementation are not disclosed, common forms of design representation are Binary Decision Diagrams (BDDs) [16] and Multiway Decision Graphs (MDGs) [17].

Two copies of the design model are built for formal analysis: the Good Machine and the Bad Machine. The same inputs and constraints are deployed on both models. Fault effects are applied in the Bad Machine only and the Strobe point of both copies are monitored. A difference in the Strobe Points indicate the propagation of the fault.

The Formal Analysis deploys formal methods to determine the Activation and Propagation of faults. Activation Analysis indicates whether the fault can be functionally activated by any combination of inputs. Propagation Analysis verifies if there is a combination of inputs that provoke fault propagation. Formal Analysis will classify the faults, which were not previously classified by the Structural Analysis, in three groups:

- Safe: Faults that cannot be activated or propagated.
- Dangerous: The tool identified at least one combination of test inputs that results in fault propagation.
- Unknown: All the others.

Formal properties to perform the analysis are automatically generated and verified with respect to all possible input stimuli. The Formal Analysis relies on formal properties and verification to prove the properties to be true.

Formal verification techniques are resource hungry and limited due to the state explosion problem. For that reason, the analysis of formal properties cannot find results for all fault targets. Therefore, the residual faults still require an alternative classification methodology.

III. Determined-Safe Faults

The ISO26262 Hardware Architectural Metrics determines the effectiveness of designs to cope with random hardware failures [18]. The failures addressed by these metrics are limited to elements that can contribute to the violation of safety goals. Safety goals define the required mitigation of hazardous events to avoid unreasonable risks caused by malfunctions. During the system development phase, safety goals will be 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 in case of faults, they may not violate safety goals. For that reason, these components can be identified by their potential to disrupt safety goals and, if applicable, determined safe.

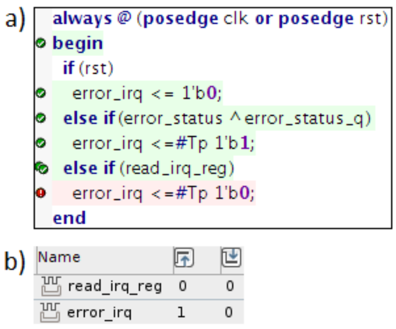
Determined-Safe Faults cannot disturb safety goals. Different from Structural-Safe Faults that cannot be tested by ANY functional test stimuli, Determined-Safe Faults may affect the output of the design. However, they cannot affect safety-critical functionalities. Common Determined-Safe fault targets are design parts not used in operational mode. The identification of these faults usually requires the judgment of hardware design experts.

A. *Determined-Safe Candidates*

In this section, we define a methodology to support the identification of Determined-Safe Faults. Our methodology deploys code coverage techniques to identify design elements that are not fully used during the design simulation. Code coverage is a method of assessing to what extent test cases exercise the design. Since this analysis relies on the simulation results, it is critical to employ representative test cases. In general, Functional Safety Verification is performed at later stages of the design life-cycle, after functional verification is completed. Therefore, we can assume that the design is available in RT and Gate level, and also comprehensive test cases are available for the identification of Determined-Safe Faults.

The initial step is to simulate the design under test, with all the available test cases, collect code coverage data, and generate the coverage reports. Our methodology does not depend on a specific tool. However, the selected tool-set should include code coverage analysis. Next, we analyze the reports to check the results for block and toggle coverage. 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 candidates for Determined-Safe Faults. For instance, by recognizing states that are never activated, as a result of block coverage, we can identify design modes that are not related to safety functionalities. Similarly, signals that are untoggled can highlight important details of the design, like invalid configurations, not utilized functions, status monitors, among others. The combination of toggle and block coverage usually provides further 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 2 illustrates an example of the correlation between the toggle and the block coverage. The block coverage (Figure 2-a) shows a block that was never activated. Since the last ”else if” statement is always false, the ’error_irq’ is not set to zero. In Figure 2-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.


a)
```
always @ (posedge clk or posedge rst)
begin
 if (rst)
  error_irq <= 1'b0;
 else if (error_status ^ error_status_q)
  error_irq <=#Tp 1'b1;
 else if (read_irq_reg)
  error_irq <=#Tp 1'b0;
end
```
b)

Name		
read_irq_reg	0	0
error_irq	1	0

Fig. 2. a) Block Coverage example - b) Toggle Coverage example.

In this example, the coverage results trigger an investigation, where we can determine that the interrupt requests (IRQ) error register is never read by the application. Next, we need to verify if this behavior is expected, and then we can decide if a fault that affects the value of the IRQ error register can be considered safe. Each candidate identified during the code coverage requires an investigation over simulation and source code. The coverage result by itself is not enough to identify the potential Determined-Safe Faults. Nonetheless, it indicates candidates that can facilitate the manual classification of such faults. After the determination of the candidates, we need to translate their behavior into a set of formal rules, allowing identification of the actual Determined-Safe Faults by Formal methods.

B. *Formal Identification of Determined-Safe Faults*

The identification of Determined-Safe Faults will deploy the same techniques described in Section II for the identification of Structural-Safe Faults. The difference is that the formal environment will incorporate the formal rules retrieved from the code coverage analysis. By constraining the environment, we enable the tool to evaluate the design in a well-specified configuration, increasing the potential for identification of Safe Faults. Additional Safe Faults will be classified as Determined-Safe, as they are Safe considering the functional constraints included in the environment.

The design elements identified during the code coverage must be translated 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. Consequently, these faults can be Determined-Safe. For example, a counter that monitors the number of transmissions is not read by the transmission controller. In that case, a failure in the monitor does not alter the design functionality. For that reason, this counter can be configured as a fault-propagation barrier, and all faults that can only affect its value can be Determined-Safe.

In most cases, the Determined-Safe Candidates translation into formal environment constraints will consider the element type. 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*. Internal signals like 'regs' and 'wires' need further analysis of the Gate-Level representation of the hardware, as they may be modified by synthesis. Nevertheless, for each environment constraint, we must confirm the assumptions by analysis of the RTL code, simulation of the design, and understanding of the expected design functionalities. An over-constrained formal environment would cause false-positives, invalidating the results.

After confirmation of the environmental constraints, we generate a file for the set-up of the Formal Analysis Tool. The set-up file must include all *assume statements* and *fault-propagation barriers*. With the set-up file in place, we repeat the formal analysis to identify the Determined-Safe Faults.

IV. Results

A. Test Case

To validate the proposed methodology, we targeted a design that is representative of the challenges of the automotive industry. For that reason, the adopted peripheral is an open hardware implementation of the SJA1000 CAN Controller, developed by Philips in the early 2000s. The selected controller implements the *BasiCAN* and the *PeliCAN* Modes. The *BasiCAN* Mode supports communication in Normal Mode with a second CAN node. The *PeliCAN* Mode supports CAN 2.0B protocol, which includes functionalities as Self-Test and Listen-Only Modes.

The test of the CAN Controller considered for this work employs a Software-Based Self-Test (SBST) approach, leading to the creation of a Software Test Library (STL). To enable the execution of the STL and emulate a realistic configuration, the CAN Controller is integrated into an OpenRISC OR1200 SoC. By deploying a full SoC, we can store the test program in a memory and control the execution of the STL during idle intervals. The complete test environment comprises two OR1200 SoCs. Each SoC is configured with a different test program and connected through a simplified version of the CAN bus avoiding the implementation of the transceiver. Instead, the resulting bus consists of the two Tx signals connected into an AND gate whose output is then connected to each Rx pin. The environment can be configured with RT or Gate level representations of the CAN Controller.

TABLE I
FAULT INJECTION RESULTS.

Fault Target	SA(1/0) Faults	Undetected Faults	Detected Faults	Diagnostic Coverage
CAN Controller	38,012	5,005	33,007	86.83%

The STL was developed as a collection of tasks that can either operate independently or collectively, depending on the self-test time slot [19]. The following tasks are available as part of the STL:

- *Bitrate Test*: aims to test the timing related modules by employing different bitrates;
- *Normal Mode Test*: tests the BasiCAN and PeliCAN Normal Modes by transmitting and receiving messages with a fixed bitrate;
- *Self-Test Mode* and *Listen-Only Mode Tests*: while one node is in Self-Test mode the other one must be in Listen-Only Mode and vice versa;
- *FIFO Test*: tests the FIFO module by filling it and emptying it while receiving several messages;
- *Errors Test*: tests error conditions due to bitrate mismatches;
- *Arbitration Loss Test*: tests arbitration loss conditions, achieved when one node stops transmitting a message due to a higher priority message being transmitted on the bus;
- *Acceptance Filter Test*: tests the acceptance filter logic that decides whether a message has to be stored in the internal memory or not.

To validate the ability of the design to cope with random hardware faults, a Fault Injection campaign was executed. We used the Cadence® Xcelium^{TM} Fault Simulator (XFS) to manage the fault campaign execution. The XFS was configured to inject SA0 and SA1 faults at every cell port of the Gate-Level representation of the CAN Controller. Table I shows the Fault Injection results. Even though the deployed STL achieves a good fault coverage (86.83%), there are still over 5,000 undetected faults. These faults must be classified to allow compliance with the requirements of ISO26262.

B. Classification of Determined-Safe Faults

During the analysis of the CAN Controller, the candidates for Determined-Safe Faults revealed some similarities. According to the intended functions, we could classify the candidates. First, several signals were constant during the simulation of the design. From those, nine are responsible for the configuration of the CAN Controller to operational

TABLE II
FORMAL ANALYSIS RESULTS.

Formal Analysis	SA(1/0) Faults	Structural Safe	Determined Safe	Total Safe
CAN Controller	38,012	539	1,996	2,535

mode. These signals were translated into *assume statements* in the constraints environment file. The combination of toggle and block coverage also revealed not used functionalities. The simulated workload does not enable modes like single-shot transmission, overload requests, and early transmission. Each of these cases must be individually analyzed. We need to define, based on the development requirements and safety goals, if these functionalities should be available in operational mode. As previously stated, our initial assumption is that the functional verification environment is available. Therefore, we can conclude that these modes are not intended in the current version of the CAN Controller. This assumption is reflected in the constraints environment file by the *assume statements* and *fault-propagation barriers*. Finally, the CAN Controller contains registers that monitor several statuses. Some of those are never read by the CPU. A misleading value in a monitor or counter that is never read by the application may not affect the expected functionality. Once again, safety goals should be verified to confirm that the CPU is not supposed to monitor these statuses. We have selected five status registers to be translated as *fault-propagation barriers* in the constraints environment file.

The constraints environment for the identification of Determined-Safe Faults on the CAN controller consisted of 10 *assume statements* and 18 *fault-propagation barriers*. We have examined the function of each included item by RTL code investigation and monitoring of the signals during the simulation. Also, some of the RTL internal signals needed to be traced to wires in the Gate level representation of the hardware to be included in the constraints environment.

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 Structural-Safe Faults. Next, we load the final constraints environment into the Formal Analysis tool and repeat step one. The additional Safe Faults identified in step two will be listed as Determined-Safe. The summary of the formal analysis results is illustrated in Table II. The computational time required for each Formal campaign was of a couple of days. As many of the properties are never proven, the total execution time depends on the timeout configured for each formal property.

C. Combined Results

The results of the Fault Injection and Formal Analysis can be combined to improve the Diagnostic Coverage. Faults that cannot disturb safety-critical functionalities, Structural and Determined-Safe, can be removed from the fault list. Each possible fault target in a design must be analyzed and classified. The annotation of the faults usually starts with Formal Analysis to identify Structural-Safe Faults. The remaining faults are simulated and, when applicable, annotated as Detected. If the desired Diagnostic Coverage is achieved, the process ends. Otherwise, the residual Undetected faults must be re-analyzed. The Determined-Safe classification is an alternative to annotate the remaining Undetected faults and increase the overall Diagnostic Coverage of the design. The Diagnostic Coverage is calculated by the formula:

$$DC = (Detected)/(Total - Safe) \qquad (1)$$

where DC is the Diagnostic Coverage, Detected are faults annotated as detected by FI Simulation, Total is the number of faults, and Safe represents the Structural and Determined-Safe Faults annotated by the Formal tool.

Figure 3 details the results of the various analysis steps. The graph illustrates the faults classification contribution achieved during Fault Injection, Structural-Safe, and Determined-Safe analysis. The process is incremental, always focusing on faults that were not previously classified. Also, Figure 3 displays the calculated Diagnostic Coverage at each step. Finally, the last column illustrates the results when all fault analyses are combined. As previously explained, we apply Formal methods to decrease the number of not classified faults. As additional Safe Faults decrease the denominator in (1), the results from the Formal analysis cause an increase in the Diagnostic Coverage.

Even with the increased fault classification, there are still Undetected faults that require further analysis. The classification of the residual faults could be achieved by improving the STL coverage, or by creating additional formal rules to increase the number of Determined-Safe Faults. The next step of our work is to propose automation techniques that can facilitate the analysis and improve even further the fault classification.

The proposed methodology appears as a promising alternative for the classification of residual faults. We define a systematic approach that allows the identification of Safe Faults based on two well-established techniques. The identification of these faults usually relies on reliability experts and requires deep knowledge over the system functionalities. This manual analysis process is strenuous and prone to errors. Our methodology is a step towards the automation of the identification of Safe Faults. By deploying the proposed methodology, we were able to classify 2,535 additional faults, resulting in a DC improvement of around 6%. With a final DC of 93.04%, the CAN Controller achieves the requirements for an automotive ASIL B hardware component as-is, i.e., without design modifications.

V. CONCLUSIONS

Functional Safety Verification is one of the most challenging steps for Integrated Circuit (IC) compliance with ISO26262. The severe demands for tolerance to random faults are a hurdle

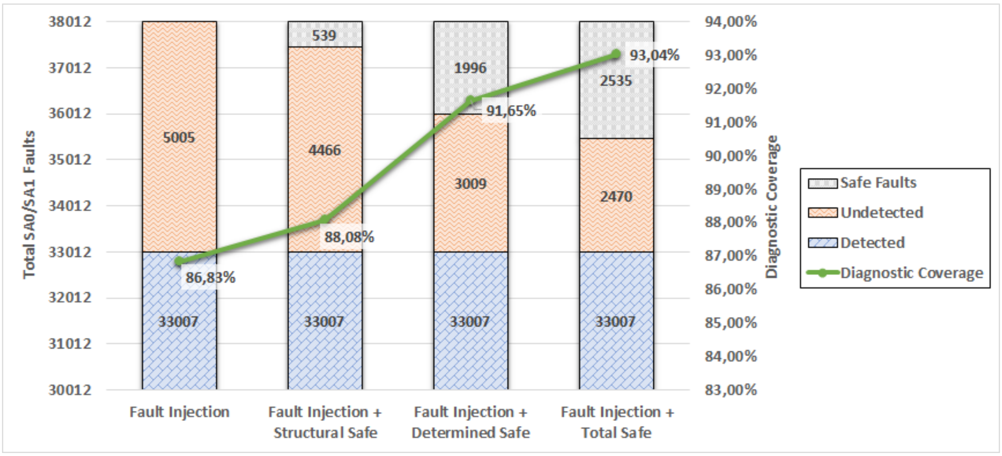


Fig. 3. Combined Results

for ICs targeting safety-critical applications. Fault analysis, as part of this process, becomes an extensive procedure that is usually repeated numerous times and requires manual inputs from specialists to achieve safety metrics. We propose a methodology that deploys code coverage and Formal analysis, as a step towards automation in Safe Faults identification. First, we identify design elements where a fault cannot disturb safety-critical functionalities. Next, those elements are translated into formal rules that are configured in a Formal analysis tool for the identification of Determined-Safe Faults. The additional classification of residual faults is necessary for compliance with ISO26262. Our methodology, in combination with Fault Simulation, was applied to a CAN Controller IP, resulting in a Diagnostic Coverage of 93%. The proposed methodology appears as a promising alternative for residual faults classification without relying solely on manual analysis.

ACKNOWLEDGMENT

This research was supported by project RESCUE funded from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

REFERENCES

[1] ARM, "Development tools and software - Software Test Libraries," https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries, 2019.
[2] Cypress, *AN204377 FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library*, 2017.
[3] Microchip, *DS52076A 16-bit CPU Self-Test Library User's Guide*, 2012.
[4] 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.
[5] 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.
[6] S. Pateras and T.-P. Tai, "Automotive semiconductor test," in *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE, apr 2017.
[7] 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.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Efficient methodology for ISO26262 functional safety verification," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, jul 2019.
[14] S. Marchese and J. Grosse, "Formal fault propagation analysis that scales to modern automotive socs," in *2017 Design and Verification Conference and Exhibition (DVCon) Europe*, 2017.
[15] A. Bernardini, W. Ecker, and U. Schlichtmann, "Where formal verification can help in functional safety analysis," in *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD*. ACM Press, 2016.
[16] G. Cabodi and M. Murciano, "BDD-based hardware verification," in *Formal Methods for Hardware Verification*. Springer Berlin Heidelberg, 2006, pp. 78–107.
[17] 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.
[18] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 5: Product development at the hardware level*, International Standardization Organization Std., Dec. 2018.
[19] R. Cantoro, S. Sartoni, and M. Sonza Reorda, "In-field functional test of CAN bus controllers," in *2020 IEEE VLSI Test Symposium (VTS) - (to appear)*. IEEE, 2020.

Appendix 9

IX

Ahmet Cagri Bagbaba, Maksim Jenihhin, Raimund Ubar, and Christian Sauer. Representing gate-level set faults by multiple seu faults at rtl. In *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2020

26th IEEE International Symposium on On-Line Testing and Robust System Design
(IOLTS)

Representing Gate-Level SET Faults by Multiple SEU Faults at RTL

Ahmet Cagri Bagbaba*†, Maksim Jenihhin†, Raimund Ubar†, Christian Sauer*
*Cadence Design Systems, Munich, Germany; † Tallinn University of Technology, Tallinn, Estonia
Email: *{abagbaba, sauerc}@cadence.com, †{maksim.jenihhin, raimund.ubar}@taltech.ee

***Abstract*—The advanced complex electronic systems increasingly demand safer and more secure hardware parts. Correspondingly, fault injection became a major verification milestone for both safety- and security-critical applications. However, fault injection campaigns for gate-level designs suffer from huge execution times. Therefore, designers need to apply early design evaluation techniques to reduce the execution time of fault injection campaigns. In this work, we propose a method to represent gate-level Single-Event Transient (SET) faults by multiple Single-Event Upset (SEU) faults at the Register-Transfer Level. Introduced approach is to identify true and false logic paths for each SET in the flip-flops' fan-in logic cones to obtain more accurate sets of flip-flops for multiple SEUs injections at RTL. Experimental results demonstrate the feasibility of the proposed method to successfully reduce the fault space and also its advantage with respect to state of the art. It was shown that the approach is able to reduce the fault space, and therefore the fault-injection effort, by up to tens to hundreds of times.**

***Index Terms*—SET, SEU, multiple faults, functional safety, hardware security, fault injection**

I. Introduction

The fault injection technique is widely used for evaluating functional safety [1] and security threats resilience [2] in integrated circuits. For safety-critical applications, it is an established, accurate method to assess the effectiveness of the deployed safety mechanisms. For security-critical applications, the technique is efficient to mimic an attack by physical fault injection aimed to alter the program flow or the processed data [3]. However, depending on the abstraction level of the circuit and the size of the fault space, a fault injection campaign can be very costly.

One of the challenges of fault injection campaigns is the vast number of possible fault locations. For a simulation-based fault injection campaign [4], engineers simulate a fault-free design and its copies with faults injected one at a time. This may imply enormous execution times, especially for the gate level fault analysis. Hence, there is a high demand for methodologies that can support designers in the early-stage design exploration of reliability factors. Moreover, fault injection into gate-level models is quite late in the integrated circuit development cycle, and any design modifications become more expensive in terms of the required engineering effort. Several researchers delved into the early-stage explorations of the designs for both safety and security applications [5]–[8]. In both safety and security-related applications, early design evaluation is necessary to minimize design iterations and resources, thus to enable faster design closure times.

In this work, we focus on SET faults at the gate level and propose an efficient solution to represent them by multiple SEU faults at the RT level. The relevance of this problem for safety-critical applications grows with the downscaling of the technology nodes, forcing designers to evaluate system's safety against SET faults, which affect combinational elements of the circuit. However, this comprehensive evaluation at the gate level is not affordable in terms of the execution time of fault injection campaigns for the industrial-sized designs. From the security point of view, SET faults at the gate level represent laser fault attacks, which can be observed in flip-flops (FFs) as single or multiple errors [9]. Here, it is crucial to evaluate laser attacks in order to determine which vulnerable SET faults create single or multiple errors in the sequential elements of the design.

To tackle the listed problems, we propose a methodology for representing gate-level transient faults, such as SETs, by Multiple Flip-Flop Upset (MFFU) at RTL. In the case of Soft Error Reliability (SER) assessment for safety applications such as automotive, MFFU becomes functionally equivalent for EDA tools to multiple simultaneous SEUs. For vulnerability analysis against fault-injection attacks on security-critical designs, MFFU refers to single and multi-bit fault injections. In this work, first, we identify static fan-in cones of each FF at the gate level. Second, we perform propagation analysis to identify SET faults that have true (sensitizable) paths to FF inputs. In this way, we obtain optimized FF sets as representatives of all SET faults to guide RTL multiple SEU fault injection campaigns. As a result, this method can successfully reduce the fault space and enhance the high complexity of fault injection campaigns. Without loss of generality, the proposed methodology is demonstrated on a Cadence EDA (Electronic Design Automation) tool flow, but it remains applicable to other tool flows as well. The main contribution of this work is as follows:

- An approach to move the gate-level SET vulnerability analysis to RTL
- A technique to reduce the fault space at RTL by applying gate-level propagation analysis
- A systematic and workload-independent methodology for representing the gate-level SETs by multiple SEUs at RTL supported by industrial-grade EDA tool flow

The rest of the paper is organized as follows. In Section II, we give an overview of the related work. The proposed

978-1-7281-8187-5/20/$31.00 ©2020 IEEE

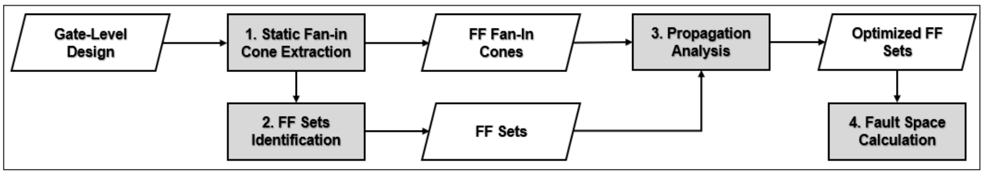

Gate-Level Design
1. Static Fan-in Cone Extraction
FF Fan-In Cones
3. Propagation Analysis
Optimized FF Sets
2. FF Sets Identification
FF Sets
4. Fault Space Calculation

Fig. 1. Steps of proposed methodology.

methodology is explained in Section III. The experimental results are discussed in Section IV. Finally, Section V concludes the paper.

II. Related Works

Relevant solutions for the above problem are proposed in [7] and [8]. However, these state-of-the-art approaches rely on the static cones pre-analysis only and do not consider if a SET fault actually propagates to the FF inputs. [7] proposes an RTL fault injection model which is representative for laser fault attacks. To do that, the authors analyse the circuits structurally and find intersection cones which guide the fault injection in advance. On the other hand, they neither create FF sets that cover all SET faults nor optimize FF sets by considering true/false paths. Similarly, [8] models the locality of a laser attack in case of multiple-bit faults. The authors analyse the circuits structurally as well and, afterwards, create FF sets. However, the authors consider only the supersets and reject all the subsets. In this way, each combination of SEUs in the superset is a trial to hit a fault in any smaller cone intersection. Yet, the probability of hitting a SET in case of any superset by selected random multiple SEU is low.

There are other studies which investigate the impact of SET faults. [10] estimates the impact of SET faults without layout information by identifying a pair of gates in which SET can propagate to multiple outputs. [11] analyzes the impact of SETs through Algebraic Decision Diagrams and Binary Decision Diagrams (BDD) and [12] improves this method by considering multiple effects. Finally, [13] suggests performing a stochastic gate-level simulation for small circuits. Last but not least, there are some works that investigate the combination of different fault analysis technologies such as [14] and [15]. These works combine the strength of formal methods and fault injection simulators; however, they analyse only permanent faults and do not analyse the representation of gate-level SET faults at RTL.

Different from the works listed above, this paper proposes a more efficient technique to prune the fault space by considering the propagation of SET faults. The significant speedup is achieved by running the RTL fault injection procedure on the accurately selected multiple flip-flop upset faults.

III. Representing Gate-Level SET Faults by Multiple SEU Faults at RTL

In this work, the aim is to identify Multiple Flip-flop Upset sets for RTL fault injection, which represent all gate-level SET faults. By doing so, we reduce the number of injections required to evaluate the effect of SET faults.

The SET fault model implies flipping the value of a signal in the combinational cloud and holding the value for a specified period of time. SEU fault model implies flipping the value of the output of a sequential element and holding it until it is overwritten with new data. SEUs can be applied on the outputs of sequential elements, such as memories, FFs and latches. We apply SET faults for one clock cycle length. The proposed flow is shown in Fig. 1 and starts with the (1) extraction of static fan-in cones of each FF in gate-level netlist. In the next step (2), FF sets are created to represent each SET faults on the fan-in cones of FFs. Then, we perform propagation analysis (3) to check if SET faults propagate to the FF inputs. If a SET fault does not propagate, then we check if this changes created FF sets. In this way, we obtain optimized FF sets, which are representative of all SET faults, which propagate to the FF inputs. Finally (4), we calculate the fault space to see the reduction when compared to state-of-the-art and random multi-bit injection approaches. The following subsections explain each step of the proposed method in detail.

A. Static Fan-in Cone Extraction of Flip-Flops at gate level

As a first step, we extract fan-in cones of each FF at the gate level, as it is illustrated in Fig. 2. In the beginning, we generate a list of all faults in the design. Then, we extract fan-in information from all FFs in the ingress combinational part of the design. Each fan-in cone search starts from a FF and expands backward, i.e. in the direction of inputs

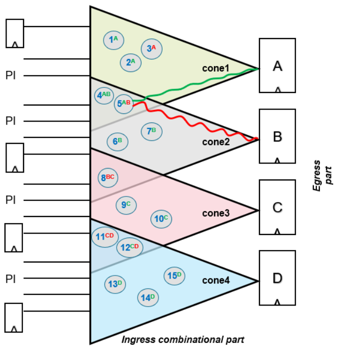


Fig. 2. Extracting fan-in cones of each FF and finding propagation paths.

TABLE I
RESULTS OF EXAMPLE DESIGN GIVEN IN FIG. 2

Affected Cone	FF Sets	Multiplicity	Optimized FF Sets	Optimized Multiplicity
Cone 1	A, B	2	A, B	2
Cone 2	A, B, C	3	A, B	2
Cone 3	B, C, D	3	C	1
Cone 4	C, D	2	D	1

of the combinational cloud until it encounters a FF output or a primary input (PI). Finally, all SETs in each cone are enumerated to map each SET to a FF set. This step is performed by using Cadence® JasperGold Functional Safety Verification App.

B. Flip-Flop Sets Identification

The second step of the proposed methodology is the identification of FF sets, which will be used as a MFFU injection target in the following steps. To do that, we consider each fan-in cone independently and determine FF sets, which cover all possible scenarios, as shown in the second column of Table I. For instance, if cone-1 is affected by a SET fault, we can cover this SET fault by injecting multiple MFFUs on A and B because cone-1 has an intersection with cone-2 which is the fan-in cone of B. This process is repeated for each cone, and FF sets are obtained with a size between 1 (in case the cone does not intersect with any other cones) and N FFs (in case all cones have an intersection).

Extracted FF sets are flip-flops of the circuit potentially affected by a SET. Therefore, MFFU injection can be limited to this set of FF. Table I also shows the multiplicity information of each FF set. The multiplicity of a FF set is the number of FF in a set. For instance, if a SET fault occurs in cone-2, it can propagate to the A, B, C FFs, causing different combinations of upsets on this set. This means that the less is the number of FF in a set (less multiplicity), the higher is the probability of hitting a real MFFU. We will use this information in the following steps. Moreover, multiplicity is important for the calculation of fault space, which will be given in the next sections. It is obvious that there are 8 combinations in one FF set with a multiplicity 3.

C. Propagation Analysis

In this work, unlike state-of-the-art researches, we also take propagation of faults into consideration in order to reduce fault space more. For this step, we deploy the formal techniques to investigate the behaviour of a design under fault. The theory behind formal techniques is creating of Boolean function representation of a design under test so that formal proves can be used. In order to achieve better performance in the modern formal tools, BDDs [16] and Multiway Decision Graphs (MDGs) [17] are widely used.

The formal analysis deploys formal methods to determine the propagation of faults. Propagation analysis verifies if there is a combination of inputs that provoke fault propagation. If a fault propagates to FF inputs, we accept that the fault has a true path to FF inputs. Otherwise, it has a false path and should be excluded from the analysis. In this step, formal properties to perform the analysis are automatically generated and verified with respect to all possible input stimuli.

The simple and high-level example in Fig. 2 illustrates that there are some SET faults in the intersection cones with a false path to the FF inputs. In this figure, green paths and superscripts point the true paths (fault propagates) while red ones show that the related fault has a false path (fault does not propagate). As a result of this step, we obtain optimized FF sets, as shown in the fourth column of Table I. It is obvious that some larger FF sets are disappeared due to non-observable faults that cannot be propagated. In this way, optimized multiplicities are obtained along with the reduced number of FF sets in some circuits. This step is performed by using Cadence® JasperGold Functional Safety Verification App. In the following subsection, we show a more detailed motivational example for the propagation analysis.

Motivational Example: Removing the paths which cannot be propagated

To explain the propagation analysis in detail, we use a motivational example given in Fig. 3 which has fan-out nodes. The circuit includes an input **x**, and outputs of the gates **AND1**, **OR1** and **OR2**. The SETs may be simulated only for these fan-outs. The steps of the approach can be listed as follows:

- Static fan-in cone analysis gives us the following FF sets of MFFU faults: (1, 2, 3, 4) for **x**, (1, 2, 3, 4) for **AND1**, (1, 2) for **OR1**, (2, 3) for **OR2**.
- After removing of duplicated sets, we get the initial sets of MFFU faults: (1, 2, 3, 4), (1, 2), (2, 3).

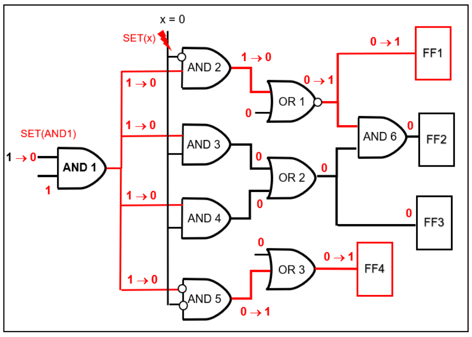


Fig. 3. Motivational example to find propagated and not-propagated faults.

- By propagation analysis, we see that for SET on AND1 we never reach all FFs, rather only either (1, 4) or (2, 3) due to the fact that the propagation of a SET at **AND1** is controlled by signal **x**=0 (by blocking two of four AND gates). Therefore, the superset (1, 2, 3, 4) for **AND1** should be replaced by subsets (1, 4) and (2, 3). In other saying, SET(**AND1**) is mapped to (1, 4) and (2, 3) FF sets.
- Moreover, the SET on the input **x** is always blocked either on **AND5** (if output of **AND1**=1), or on **AND2**, **AND3**, **AND4** (if output of **AND1**=0). Hence, the superset (1, 2, 3, 4) for SET(x) should be replaced by (1, 2, 3).
- As a result, we get instead of initial (1, 2, 3, 4), (1, 2), (2, 3), optimized FF sets (1, 2), (2, 3), (1, 4) (2, 3), (1, 2, 3), where (2, 3) can be removed as it is duplicated.
- Thus, the final optimized FF sets: (1, 2), (2, 3), (1, 4), (1, 2, 3).

In this motivational example, we analyzed the propagation of SETs only on **x** and the outputs of **AND1**, **OR1** and **OR2**. The propagation analysis is sufficient for the SETs at these four locations that also represent the remaining SET faults in the fan-out free regions.

D. *Fault Space Calculation*

In the fault injection procedure, SEUs are injected in all possible locations and at each clock cycle [18]. Therefore, the number of injections required for a single transient fault is large, especially for the industrial-sized designs. When considering the size and low speed of fault injection simulations at the gate level, optimization methods should be applied. Hence, considering the huge number of SET injections at the gate level, our proposed method significantly reduces the number of injections by identifying optimized FF sets when compared to state-of-the-art and random multi-bit injection approaches applied in safety and security applications.

Our proposed methodology can significantly reduce the fault space by leveraging the FF sets with propagation analysis. In this work, we compare our results with the state-of-the-art and random multi-bit injection. State-of-the-art researches such as [7] and [8] rely on only a static approach and do not consider the propagation analysis. Similarly, the random multi-bit injection method considers all possible FF combinations. In order to calculate fault space or the number of injections, we use the following equation where N is the number of FFs, k_1, k_2, ..., k_N are the numbers of FF in each set and $1 \leq k_i \leq N$. given in [8].

$$FaultSpace_{Total} = \sum_{i=1}^{N} (2^{k_i} - 1) \qquad (1)$$

By using the above equation, the total fault space for the example given in Fig. 3 can be calculated effortlessly. As it is explained in Section III-C, we have the initial and not-optimized sets which represent the state-of-the-art approach as (1, 2, 3, 4), (1, 2) and (2, 3). By using the given formula, the total number of faults is 21. On the other hand, we have optimized FF sets as (1, 2), (2, 3), (1, 4), (1, 2, 3), which require 16 number of injections. Therefore, our proposed method can reduce the total fault space from 21 to 16 for the motivational example given in Fig. 3.

IV. Experimental Results and Discussion

In order to verify the effectiveness of proposed methodology, we evaluate our methodology on the ITC'99 [19] benchmark circuits.

In order to perform fan-in cone analysis and propagation analysis, we deploy Cadence tools along with the developed script sets, which execute on gate-level design. Meanwhile, all applied methods remain applicable to other tool flows. In the beginning, we synthesize Verilog or VHDL design through Cadence® Genus™ Synthesis Solution to obtain gate-level representation of the design. Then, steps 1, 2 and 3 shown in Fig. 1 are performed on our application which deploys Cadence® JasperGold Functional Safety Verification App.

We use three methods to show the fault space reduction and compare the results. The first method is "without propagation analysis" which represents the state-of-the-art as in [8]. The main difference between our proposed methodology "with propagation analysis" and the state-of-the-art is the identification of true (sensitizable) paths. We leverage the analysis by identifying SET faults which do not propagate to FF inputs so that fault space is reduced more. In other words, we cut down the pessimism in the results. The third approach used for comparison is "Random Multi-Bit injection". This is basically injecting faults on all possible combinations of FFs randomly that naturally causes huge fault space. Our application is capable of building the fault space for each method and given design without any significant effort.

All experimental results are presented in Table II. The selected designs include various designs from the ITC'99 benchmark. During creating of FF sets, we remove faults on clock and reset signals from the analysis due to the fact that the clock tree is not known in this stage of the design. Other faults except clock and reset are kept as they are. This step is done in our application automatically. We show the number of sets, number of supersets, maximum multiplicity and calculated fault spaces for each analysis and design. The number of sets shows the number of all identified FF sets before duplicated ones are removed. In contrast, the number of supersets points the same after duplicated ones are removed. Total Faults are calculated by using the Equation 1.

In Table II, it can be seen that our proposed methodology reduces the Total Faults significantly when compared to both state-of-the-art and the random multi-bit injection approaches. For some circuits such as **b01** and **b08**, we are able to reduce only the number of supersets while the maximum multiplicity is still the same in both cases. Moreover, there is no optimization achieved in **b06**. For the rest of the circuits given in Table II, we both optimize the number of supersets and maximum multiplicity. Thereby, the total set of faults are optimized significantly, as shown in Fig. 4 (values are normalized). It is observable that total faults in the proposed methodology (orange bars) are less than the other

TABLE II
EXPERIMENTAL RESULTS: FAULT SPACES ACHIEVED BY THREE METHODS

Circuit	# FF	without propagation analysis				with propagation analysis				Random Multi-Bit Injection
		# sets	# superset	max multiplicity	Total Faults	# sets	# superset	max multiplicity	Total Faults	Total Faults
b01	5	5	2	4	1.80E+01	3	1	4	1.50E+01	3.20E+01 - 1
b02	4	4	1	3	7.00E+00	1	1	2	3.00E+00	1.60E+01 - 1
b03	30	8	3	12	4.14E+03	3	1	9	5.11E+02	1.07E+09 - 1
b04	66	27	10	19	4.00E+06	5	4	8	1.02E+03	7.38E+19 - 1
b05	34	62	2	33	9.00E+09	61	5	31	2.00E+09	1.72E+10 - 1
b06	8	7	5	4	4.30E+01	7	5	4	4.30E+01	2.56E+02 - 1
b07	46	51	2	35	4.00E+10	43	3	26	8.00E+07	7.04E+13 - 1
b08	21	19	2	18	2.70E+05	11	2	18	2.62E+05	2.10E+06 - 1
b09	28	14	1	28	3.00E+08	7	1	27	1.00E+08	2.68E+08 - 1
b10	17	45	9	11	5.91E+03	13	4	11	2.62E+03	1.31E+05 - 1
b11	31	43	9	18	4.65E+05	9	2	16	6.60E+04	2.15E+09 - 1
b13	50	40	13	13	9.15E+03	20	9	9	9.47E+02	1.13E+15 - 1

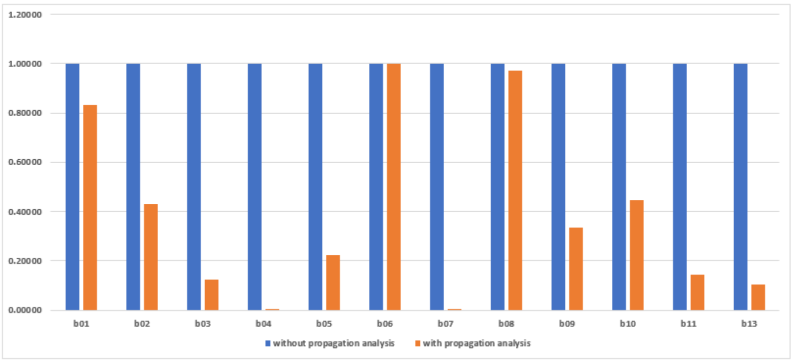


Fig. 4. Fault Space comparison.

two methods. We also add that we reduce the fault space from 1.20 times to a few hundred times when compared without propagation analysis, depending on the circuit.

Moreover, we also compare our results with the well-known Statistical Fault Injection (SFI) approach [20] in case initial population sizes calculated before are used. SFI can be used for transient fault injection campaigns to reduce the execution times while keeping a meaningful number of injections with an error margin. This is one of the possible ways to perform RTL fault injection campaigns after FF sets are defined by using the methodology presented in this paper. In an SFI campaign, the sample size or the margin of the error with a certain confidence level are determined by using the Equation 2 defined in [20]. In this way, it is possible to obtain precise results while injecting a small number of faults [20]. The technique allows to know the margin of error while restricting the campaign time to the minimum. To sum up, there are three confidence levels in SFI as 90%, 95%, and 99.8%. In this work, we only use the 95% confidence level as it is the one that is practically used in the industry. Also, three error margins are defined as 5%, 1% and 0.1%.

$$n = \frac{N}{1 + e^2 \times (\frac{N-1}{t^2 \times p \times (1-p)})} \tag{2}$$

In Table III, we show the SFI results. In this table, N shows the initial population. In our case, N is equal to the total faults shown in Table II. Moreover, n(5%), n(1%) and n(0.1%) show the required sample size with the error margins 5%, 1% and 0.1% respectively. This shows that our proposed methodology can prune the fault space from 1.12 times to a few hundred times in case faults are injected by using SFI. Note, the results for some sample sizes remain similar due to the fact that the initial population is always finite. Even so, we show that a significant reduction is achieved by using the proposed methodology, especially when we reduce the error margins. Therefore, it is efficient to use the proposed methodology and to select a sample for fault injection among the pre-defined initial populations in the MFFU space identified using the method ”with propagation analysis”.

V. CONCLUSIONS

In this work, we propose a methodology to represent gate-level SET faults by multiple SEU faults at RTL. It enables a solution for the high complexity problem of expensive gate-level fault injection campaigns by changing the abstraction level. We improve the state-of-the-art by considering propagation analysis of each SET fault. First, we find static fan-in cones of each FF at the gate level. Second, FF sets are

TABLE III
COMPARISON OF THREE METHODS IN A SFI CAMPAIGN WITH 95% CONFIDENCE LEVEL

Circuit	**without propagation analysis**				**with propagation analysis**				**Random Multi-Bit Injection**			
	N	**n(5%)**	**n(1%)**	**n(0.1%)**	**N**	**n(5%)**	**n(1%)**	**n(0.1%)**	**N**	**n(5%)**	**n(1%)**	**n(0.1%)**
b01	1.80E+01	1.70E+01	1.80E+01	1.80E+01	1.50E+01	1.40E+01	1.50E+01	1.50E+01	3.20E+01 - 1	3.00E+01	3.20E+01	3.20E+01
b02	7.00E+00	7.00E+00	7.00E+00	7.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	1.60E+01 - 1	1.50E+01	1.60E+01	1.60E+01
b03	4.14E+03	3.52E+02	2.89E+03	4.12E+03	5.11E+02	2.20E+02	4.85E+02	5.11E+02	1.07E+09 - 1	3.84E+02	9.60E+03	9.60E+05
b04	4.00E+06	3.84E+02	9.58E+03	7.74E+05	1.02E+03	2.79E+02	9.22E+02	1.02E+03	7.38E+19 - 1	3.84E+02	9.60E+03	9.60E+05
b05	9.00E+09	3.84E+02	9.60E+03	9.60E+05	2.00E+09	3.84E+02	9.60E+03	9.60E+05	3.44E+10 - 1	3.84E+02	9.60E+03	9.60E+05
b06	4.30E+01	3.90E+01	4.30E+01	4.30E+01	4.30E+01	3.90E+01	4.30E+01	4.30E+01	2.56E+02 - 1	1.54E+02	2.49E+02	2.56E+02
b07	4.00E+10	3.84E+02	9.60E+03	9.60E+05	8.00E+07	3.84E+02	9.60E+03	9.49E+05	7.04E+13 - 1	3.84E+02	9.60E+03	9.60E+05
b08	2.70E+05	3.84E+02	9.28E+03	2.11E+05	2.62E+05	3.84E+02	9.27E+03	2.06E+05	2.10E+06 - 1	3.84E+02	9.56E+03	6.59E+05
b09	3.00E+08	3.84E+02	9.60E+03	9.57E+05	1.00E+08	3.84E+02	9.60E+03	9.51E+05	2.68E+08 - 1	3.84E+02	9.60E+03	9.57E+05
b10	5.91E+03	3.61E+02	3.66E+03	5.88E+03	2.62E+03	3.35E+02	2.06E+03	2.62E+03	1.31E+05 - 1	3.83E+02	8.95E+03	1.15E+05
b11	4.65E+05	3.84E+02	9.41E+03	3.14E+05	6.60E+04	3.82E+02	8.39E+03	6.18E+04	2.15E+09 - 1	3.84E+02	9.60E+03	9.60E+05
b13	9.15E+03	3.69E+02	4.69E+03	9.06E+03	9.47E+02	2.74E+02	8.62E+02	9.46E+02	1.13E+15 - 1	3.84E+02	9.60E+03	9.60E+05

created pessimistically, meaning that propagation analysis is not considered. Third, we execute propagation analysis by using a formal approach to find SET faults that propagate to FF inputs. Then, optimized FF sets are created again with less pessimism. Finally, we calculate the fault space to show the effectiveness of the proposed methodology. In this way, we significantly reduce the number of fault injections and obtain a higher probability of hitting a true multiple SEU fault. Experimental results show that we make the fault space smaller by up to tens to hundreds of times.

As future work, we aim to apply this methodology for functional safety and security evaluation in industrial-sized CPU designs.

ACKNOWLEDGMENT

This research was supported by project RESCUE funded from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowaska-Curie grant agreement No 722325.

REFERENCES

[1] B. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Speeding up safety verification by fault abstraction and simulation to transaction level," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Sep. 2016, pp. 1–6.

[2] R. Leveugle, "Early analysis of fault-based attack effects in secure circuits," *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1431–1434, Oct 2007.

[3] N. Wiersma and R. Pareja, "Safety != security: On the resilience of asil-d certified microcontrollers against fault injection attacks," in *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Sep. 2017, pp. 9–16.

[4] H. Ziade, R. Ayoubi, and R. Velazco. A Survey on Fault Injection Techniques. (2003). [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.167.966

[5] C. Kumar, F. Maamari, K. Vittal, W. Pradeep, R. Tiwari, and S. Ravi, "Methodology for early rtl testability and coverage analysis and its application to industrial designs," in *2014 IEEE 23rd Asian Test Symposium*, Nov 2014, pp. 125–130.

[6] S. Mirkhani and J. A. Abraham, "Eagle: A regression model for fault coverage estimation using a simulation based metric," in *2014 International Test Conference*, Oct 2014, pp. 1–10.

[7] A. Papadimitriou, D. Hély, V. Beroulle, P. Maistri, and R. Leveugle, "A multiple fault injection methodology based on cone partitioning towards rtl modeling of laser attacks," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–4.

[8] P. Vanhauwaert, P. Maistri, R. Leveugle, A. Papadimitriou, D. Hély, and V. Beroulle, "On error models for rtl security evaluations," in *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, May 2014, pp. 1–6.

[9] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid, "Modular fault injector for multiple fault dependability and security evaluations," in *2011 14th Euromicro Conference on Digital System Design*, Aug 2011, pp. 550–557.

[10] D. Rossi, M. Omana, F. Toma, and C. Metra, "Multiple transient faults in logic: an issue for next generation ics?" in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, Oct 2005, pp. 352–360.

[11] N. Miskov-Zivanov and D. Marculescu, "Mars-c: modeling and reduction of soft errors in combinational circuits," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 767–772.

[12] N. Miskov-Zivanov and D. Marculescu, "Multiple transient faults in combinational and sequential circuits: A systematic approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 10, pp. 1614–1627, Oct 2010.

[13] A. Mochizuki, N. Onizawa, A. Tamakoshi, and T. Hanyu, "Multiple-event-transient soft-error gate-level simulator for harsh radiation environments," in *TENCON 2015 - 2015 IEEE Region 10 Conference*, Nov 2015, pp. 1–6.

[14] 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)*, Dec 2019, pp. 129–1295.

[15] F. A. d. Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Efficient methodology for iso26262 functional safety verification," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, July 2019, pp. 255–256.

[16] G. Cabodi and M. Murciano, "Bdd-based hardware verification," in *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 78–107.

[17] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny, "Multiway decision graphs for automated hardware verification," 1996.

[18] A. C. Bagbaba, M. Jenihhin, J. Raik, and C. Sauer, "Accelerating transient fault injection campaigns by using dynamic hdl slicing," in *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, Oct 2019, pp. 1–7.

[19] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level itc'99 benchmarks and first atpg results," *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, July 2000.

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.

Appendix 10

X

Ahmet Cagri Bagbaba, Felipe Augusto da Silva, Matteo Sonza Reorda, Said Hamdioui, Maksim Jenihhin, and Christian Sauer. Automated identification of application-dependent safe faults in automotive systems-on-a-chips. *Electronics*, 11(3), 2022

electronics

Article

Automated Identification of Application-Dependent Safe Faults in Automotive Systems-on-a-Chips

Ahmet Cagri Bagbaba [1,2,*], Felipe Augusto da Silva [1,3], Matteo Sonza Reorda [4], Said Hamdioui [3], Maksim Jenihhin [2] and Christian Sauer [1]

[1] Cadence Design Systems, 85622 Munich, Germany; dasilva@cadence.com (F.A.d.S.); sauerc@cadence.com (C.S.)
[2] School of Information Technologies, Department of Computer Systems, Tallinn University of Technology, 19086 Tallinn, Estonia; maksim.jenihhin@taltech.ee
[3] Mathematics and Computer Science, Department of Quantum and Computer Engineering, Faculty of Electrical Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands; s.hamdioui@tudelft.nl
[4] Department of Control and Computer Engineering, Politecnico Di Torino, 10129 Torino, Italy; matteo.sonzareorda@polito.it
* Correspondence: abagbaba@cadence.com

Abstract: ISO 26262 requires classifying random hardware faults based on their effects (safe, detected, or undetected) within integrated circuits used in automobiles. In general, this classification is addressed using expert judgment and a combination of tools. However, the growth of integrated circuit complexity creates a huge fault space; hence, this form of fault classification is error prone and time consuming. Therefore, an automated and systematic approach is needed to target hardware fault classification in automotive systems on chips (SoCs), considering the application software. This work focuses on identifying safe faults: the proposed approach utilizes coverage analysis to identify candidate safe faults considering all the constraints coming from the application. Then, the behavior of the application software is modeled so that we can resort to a formal analysis tool. The proposed technique is evaluated on the AutoSoC benchmark running a cruise control application. Resorting to our approach, we could classify 20%, 11%, and 13% of all faults in the central processing unit (CPU), universal asynchronous receiver–transmitter (UART), and controller area network (CAN) as safe faults, respectively. We also show that this classification can increase the diagnostic coverage of software test libraries targeting the CPU and CAN modules by 4% to 6%, increasing the achieved testable fault coverage.

Keywords: automotive systems; fault classification; fault injection; formal methods; functional safety; diagnostic coverage; ISO 26262; safe faults

check for updates

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

Academic Editor: Patrick Siarry

Received: 18 December 2021
Accepted: 17 January 2022
Published: 20 January 2022

1. Introduction

Complex hardware and software systems are frequently used in safety critical environments such as automobiles, planes, or medical devices. Safety standards have been introduced to estimate and reduce the risk of critical failures in embedded systems utilized in these areas. This risk might correspond to physical injury or damage to the overall health of humans. Therefore, special solutions for hazards mitigation are required to develop systems working in critical domains. Industries in the above domains need to comply with standards focusing on the development of hardware/software components according to system requirements [1]. Concerning the automotive industry, the number of systems on chip (SoCs) and applications deployed in automobiles is significantly increasing with the final objective of developing self-driving cars. Modern automobiles already incorporate more than 100 electronic control units (ECUs) [2] to cope with the challenges originating from complex applications, such as advanced driver assistance systems (ADAS). Complexities of the hardware and software applications escalate on both the architectural and functional levels. The hardware complexity is defined as how many components and blocks are integrated on a single SoC/chip. The software complexity is related to the number and time complexity of the pieces that should be combined to deliver the functionality and internal interactions. Moreover, migrating to more advanced integrated circuit (IC) technologies poses a more significant challenge for the safety of automobiles since several phenomena, such as nanoelectronics aging, process variation, or electrostatic discharge used in advanced nodes, introduce numerous vulnerabilities [3]. Consequently, the automotive industry has developed the ISO 26262 Road Vehicles Functional Safety Standard [4] to minimize the risks connected to electric and/or electronic systems used in vehicles. For automotive applications, each electronic system must detect and correctly manage a high percentage of potential faults during the operation in the field to avoid life-critical situations. In order to decide which faults could disturb the safety critical functionality of an IC, faults must be classified based on their effects in the operation mode using expert judgment and a combination of tools. From this perspective, faults can be classified as safe or dangerous. A safe fault does not contribute to the violation of the safety goal, whereas a dangerous fault may lead to a failure relevant for the overall system, that is, create a hazard. We note that all the terms and definitions are given in the context of functional safety verification guided by ISO 26262. Examples of safe faults include faults located in parts of an IC that are not used by the application and faults masked by some safety mechanism. Fault classification is of prime importance for the test of ICs in the operational mode. This test can be performed resorting to different solutions, including design for testability (e.g., BIST) and software test libraries (STLs) based on the software-based self-test (SBST) paradigm [5]. In both cases, the identification of safe faults is vital since it enables us to remove safe faults from the initial (normally huge) fault list and to focus the test efforts toward the remaining faults, i.e., the testable ones [6]. Identifying safe faults thus makes it easier to reach the target diagnostic coverage (DC), helping to achieve safety requirements, such as a higher automotive safety integrity level (ASIL) [4]. For these reasons, there is a high demand for an automated, systematic, and comprehensive safe fault identification technique.

The effects of a fault classification flow are summarized in Figure 1, referring to a generic case study. We assume that an SoC runs a single software (SW) application during its operational life and uses an STL as a safety mechanism. Therefore, the DC of this STL must be calculated to prove that it detects dangerous faults up to a certain extent in the target design. In the first step of the flow, without any classification, all the faults are unknown, as shown in Figure 1. Then, an initial classification is performed to identify the first group of structurally safe faults, i.e., those which are safe due to the IC structure (e.g., faults located on lines which are not connected to the IC primary inputs and/or outputs). These kinds of safe faults can be identified using any automatic-test-pattern-generation (ATPG) or formal analysis tool. However, other safe faults may exist, which cannot be identified by these tools; therefore, a considerable amount of faults are still unknown after the first step. The unknown faults need to be further analyzed to check whether their effects may impact the safety critical functionalities or not. Thus, fault simulation with an STL is deployed to classify faults better. In practice, this step (named unoptimized classification in Figure 1) produces inaccurate results since it is often impossible to exhaustively evaluate all possible input stimuli or activate all possible operating modes in an application or system [7]. Undetected faults may correspond either to safe or dangerous faults. As in Figure 1, fault simulation targets unknown faults and classifies them as either detected or undetected based on the propagation of faults. A non-negligible amount of undetected faults may be observed depending on the workload that runs on the target design. Usually, all the undetected faults are pessimistically classified as dangerous. For this reason, the gathered figures from fault simulation may not be representative of the design operational behavior, as not all faults can be accurately classified. DC is calculated in this step using (1), where Detected is the number of faults classified as detected and dangerous by fault simulation; Total is the size of the target system's fault list; and Safe is the number of safe faults. The purpose is to check if the collected results from fault simulation satisfy the desired safety metrics. If DC is not enough, the test must be improved, or an additional classification effort targeting undetected faults, i.e., a subset of undetected faults, is required to classify their effects. Experts usually perform this step based on their design knowledge; however, this is error prone and time consuming. Consequently, the unoptimized classification implies that there is still room for improvement in the fault classification pessimism. Finally, using the technique presented in this work, a formal analysis approach optimizes the fault classification (named optimized classification) as shown in the fourth bar of Figure 1, which targets the identification of more safe faults, reducing the number of undetected faults and, therefore, the overall pessimism of the classification. The optimized classification decreases the denominator of (1) by classifying more safe faults than in the unoptimized classification, and the DC is increased.

$$DC = Detected/(Total - Safe) \tag{1}$$

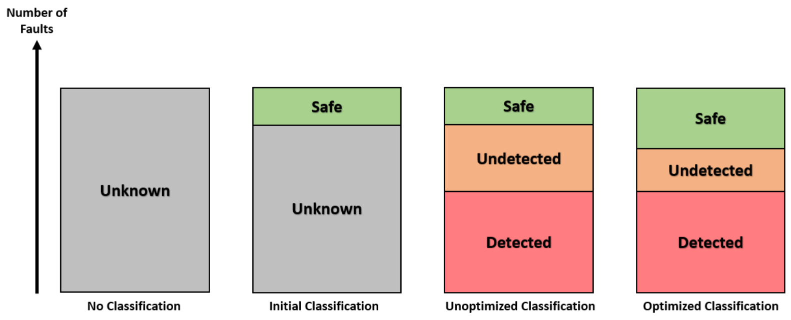


Figure 1. Hardware fault classification flow.

This work advances hardware fault classification with an automated workflow, which assists safety experts in addressing fault classification reducing human error and the time to signoff. The present work focuses on the automated analysis of undetected faults to check whether they affect the safety critical functionalities of ICs. In the case that a fault cannot violate a safety goal or disturb safety critical outputs, it is defined as a safe fault. We consider a realistic scenario corresponding to a special-purpose system, i.e., an SoC which performs a single SW application, which remains the same during the whole operational life. Using the proposed technique, we can identify application-dependent safe (App-Safe) faults. One example of App-Safe faults is associated with the faults in the CPU debug unit, which is not used by the SW application during the operation life of the SoC. For this purpose, first, we perform several logic simulations to extract a target system's operational behavior by investigating code coverage results. Then, the candidates for being labeled safe faults which are not safety related are automatically translated into formal properties, which then configure the formal environment to identify App-Safe faults.

As a case study, the AutoSoC benchmark [8], an automotive representative SoC, and the cruise-control-application (CCA) as a target SW application are used. We focused on the CPU core and several peripherals, i.e., the universal asynchronous receiver–transmitter IP (UART) and the controller area network controller IP [9] (hereinafter referred to as CAN).

This paper addresses the problem of what is new in ISO 26262 functional safety verification that differs from general reliability in terms of safe faults. The main goal of functional safety verification is to avoid safety goal violations, not general failures in the design. This is the concept of safe faults. Our hypothesis is on deploying the strengths of existing technologies in an innovative methodology to resolve the issues. As a result, this paper proposes a novel methodology based on the innovative use of existing technologies that address the problem. The main contributions of this work can be listed and summarized as follows:

- A new systematic approach combined with engineering concepts in order to deliver an industrial solution that can be deployed for SoC targeting the automotive industry.
- An automated safe fault identification technique supported by an industrial-grade electronic design automation (EDA) tool flow: logic simulation of the target design when it runs the software application, extraction of coverage reports that reflects the behavior of the software application, development of formal properties that are translated from coverage reports, and formal analysis execution.
- ISO 26262-driven safe fault identification technique that contributes to the testing and verification theory by focusing the test efforts on the other faults (dangerous).
- A scalable formal property generation approach to translate the design's operational behavior into the formal analysis tool.
- An experimental demonstration of the effectiveness of the proposed technique on a comprehensive automotive benchmark SoC, using its CPU and the UART and CAN peripherals.
- Significant improvements in the classification of safe faults and of the resulting DC, thus allowing to achieve a higher safety level. When the AutoSoC runs the CCA, 20%, 11%, and 13% of all faults in the CPU, UART, and CAN are classified as safe using the presented technique, respectively. The value of DC is increased by around 6% and 4% for the CPU and the CAN, respectively. This analysis also reduces the number of undetected faults by 1.5 and 1.6 times in the CPU and CAN, respectively.

The rest of this paper is structured as follows. Section 2 summarizes the previous and related works in the area. Section 3 provides some background, covering hardware fault classification and the techniques to achieve this classification, such as fault simulation and formal analysis. Section 4 defines the App-Safe faults in detail and presents the proposed method step by step. Section 5 briefly describes the AutoSoC benchmark suite, including its CPU, peripherals, and the software application that we use in this work. Section 6 reports and discusses the experimental results of the proposed technique. Finally, Section 7 draws some conclusions.

2. Related Works

Many works exist in the literature about hardware fault classification. This section examines some of them based on different approaches, such as fault simulation, formal methods, ATPG, or hybrid approaches.

Several works have explored fault simulation targeting fault classification. For example, Ref. [10] optimizes fault simulation by integrating it into the design verification environment and using the clustering approach to accelerate the fault simulation campaigns. However, using only fault simulation for fault classification is computationally expensive and incomplete; hence it requires additional methods to classify undetected faults. Similarly, Ref. [11] relies only on fault simulation to classify the faults, but there was no additional classification technique proposed. Similarly, Refs. [12–15] deploy fault simulation to classify faults in automotive systems considering the requirements of ISO 26262. In short, when fault simulation is used alone to classify faults, additional techniques targeting the classification of undetected faults are necessary.

Hence, some other works have investigated formal analysis, focusing on safe fault identification. Refs. [16–18] use the ability of the formal techniques to analyze the design behavior. Safe fault identification is also applied to GPUs. For example, ref. [19] employs formal analysis to increase fault coverage when the identification technique is applied to an open-source GPU. These works specifically focused on identifying structurally safe faults, i.e., faults for which there are no test or input stimuli due to the hardware structure, independently of the software and the application.

Researchers have also combined fault simulation and formal analysis leveraging fault classification. Refs. [20,21] have an eclectic approach that makes use of the strength of different technologies. Even though these works are promising in terms of the results, they still require many manual efforts based on the engineer's expertise.

On the other side, ATPG is also a promising technique to identify safe faults. Examples of this approach are [22–24], which aim at identifying untestable faults in sequential circuits. We note that untestable faults are, by definition, safe faults [25]. In addition, Refs. [6,25,26] resort to ATPG to identify application-dependent safe faults, which is the same target of the work described in this paper. Even though these works can identify safe faults using the ATPG, they still have a manual part in their flow, i.e., they are semi-automated.

Considering application-dependent safe faults, some works have proposed solutions for the classification of these kinds of faults. For example, Ref. [27] explores the use of safe faults to optimize STL fault coverage in microprocessors, which is not safety critical. However, the scope of the work is limited only to CPU modules, and the deployed tests are not automotive representative. Additionally, Ref. [28] focuses on safe fault identification in only CAN; thus, the analysis of a complete automotive representative SoC is missing. In addition, Ref. [28] analyzes a combination of test programs developed for CAN, which makes it weaker as this work examines safe faults when an SoC runs a practical industry-scale software application. Last, the presented work in this paper has a more advanced approach in the sense that the proposed technique is more automated and systematic; hence, it is less error prone and time consuming.

To address the outlined gaps, the technique proposed in this paper corresponds to a fully automated fault classification technique, which focuses on safe faults when a CPU is running a specific SW application. The main strength of the proposed approach lies in the developed formal properties, which are extracted via the analysis of the target system's operational behavior.

3. Background

This section, first, provides basics about hardware fault classification. Then, fault simulation and formal methods for hardware fault classification are explained.

3.1. Hardware Fault Classification

ISO 26262 divides the malfunction of electrical/electronic components into two categories, corresponding to systematic and random faults [4]. A systematic fault is manifested in a deterministic way and can only be prevented by applying process or design measures. On the other hand, a random fault can occur unpredictably during the lifetime of a hardware element. When we consider safety critical designs, such as automotive, medical, or aerospace designs, safety and verification engineers must prove that both the correct and safe functionalities of these designs are guaranteed, taking into account both systematic and random faults.

Several sources exist for random hardware faults, such as extreme operating conditions, aging, or in-field radiation. Additionally, each fault type should have a fault model that describes how faults from these sources should be modeled at the appropriate hardware design abstraction level (e.g., at the gate level or register-transfer level (RTL)). Moreover, faults can be permanent and transient. Transient faults occur and subsequently disappear. On the other hand, permanent faults occur and stay until removed or repaired. This work focuses on permanent faults modeled as stuck-at faults, i.e., signals getting permanently stuck at a given logic value, i.e., 0 (stuck-at-0, SA0) or 1 (stuck-at-1, SA1), following what safety standards in the automotive domain suggest. We also note that a stuck-at fault can apply to all netlist signals, such as the ports of logic gates or registers. In this paper, we focus on random hardware faults (specifically permanent faults), only.

In order to determine the probability of a fault causing a safety critical failure, its effects must be classified into two different categories as follows.

- Safe: A safe fault does not disturb any safety critical functionality because it is not in safety relevant logic, or it is in safety relevant logic but is unable to impact the safety critical functionality of a design (i.e., it cannot violate a safety goal).

- Dangerous: A dangerous fault impacts the safety of the device and creates a hazard that may produce a safety goal violation.

3.2. Fault Simulation

As an integral part of the safety critical IC development, fault simulation is a widely used technique to identify fault effects [29]. Fault simulation tools analyze an RTL or gate-level abstraction of an IC by performing a simulation with some given test stimuli. In general, the fault injection flow is based on the comparison between the results of the good run and those of the faulty run. First, the good run is run to generate reference values. In this step, observation points where the propagation of faults is monitored are specified. Then, the faulty run is executed with faults injected. In the end, the reference values obtained by the good run and the faulty values generated by the faulty run are compared for the classification of each injected fault, and we can determine whether each injected fault is detected or undetected. Faults are classified as detected when at least one output value changes for a specified observation point between the good run and the faulty run. Otherwise, the fault is classified as undetected.

Although fault simulation is a widely used and adopted technique by both industry and academia, it suffers from two problems [7]:

- Incomplete results: It is impossible to simulate all possible combinations of input sequences when considering today's complex applications and devices. Hence, some faults cannot be accurately classified as safe or dangerous with the fault simulation technique.
- No-effect faults: Faults injected into components of the target system that are not activated during the execution of a workload (testbench) will result in no-effect faults. These faults are classified as undetected by the fault simulation. This causes ambiguous results because these faults might be dangerous when different or more comprehensive input stimuli are used.

Because of the two reasons listed above, it may be required to use additional classification techniques, such as formal methods, as explained in the following subsection, to classify faults after fault simulation, whether they are safe or not. We must also mention that both sets of detected and undetected faults may contain safe and dangerous faults. Therefore, if a fault is not classified and not proven safe, it should be pessimistically considered dangerous.

The following subsection explains how formal methods classify faults, distinguishing between safe and dangerous.

3.3. Formal Methods

Formal methods help to classify faults based on their effects. An analysis is performed to determine whether or not a target design satisfies a set of properties or conditions. This approach is usually a combination of different techniques that employ static analysis and algorithmic calculations. Compared to fault simulation that applies one single stimulus, formal analysis is less limited since it abstracts from any specific stimulus. On the other hand, the computational complexity may limit the formal analysis applicability [28]. In this case, the classification of all faults can be impossible; thus, a formal analysis tool should be fed by formal properties, developed carefully, considering the constraints from the SW application and looking for a compromise between computational feasibility and result accuracy, as it is done in this work.

In general, formal tools apply two checks, structural analysis check and formal analysis check to identify safe faults, as explained below.

3.3.1. Structural Analysis Check Types

In the structural analysis check, formal tools use the topological characteristics of a design to determine the testability of each fault. There are three methods of structural fault analysis:

- Out-of-cone of influence (COI) analysis: This method checks whether a given node is outside the COI of a given observation point(s); in that case, the fault is safe. In Figure 2, all faults located on nodes in the COI of out_1 (shown in green) are safe since the considered observation point is out_0 in the example analysis. It is obvious that stuck-at faults on the cell ports of G3 cannot propagate to out_0, as they have no physical connection with out_0. Hence, faults on G3 are safe.
- Unactivatable analysis: This is to check if a SA0 or SA1 fault is located on a node that is constant 0 or 1; if so, the fault cannot be activated. In this case, the fault is unactivatable and safe. In Figure 3, assuming that in_0 is tied to logic zero, f_0 for SA0 is unactivatable and safe.
- Unpropagatable analysis: This is performed to check if a fault is activated and in the COI of the considered observation point but cannot be propagated to the outputs. In this case, the fault is safe. In Figure 3, the AND gate G2 can block the propagation of f_1 if one of the in_1 or in_2 is always set with the logic value zero. Hence, f_1 would be safe for SA1 or SA0, as it can never be propagated to out_0.

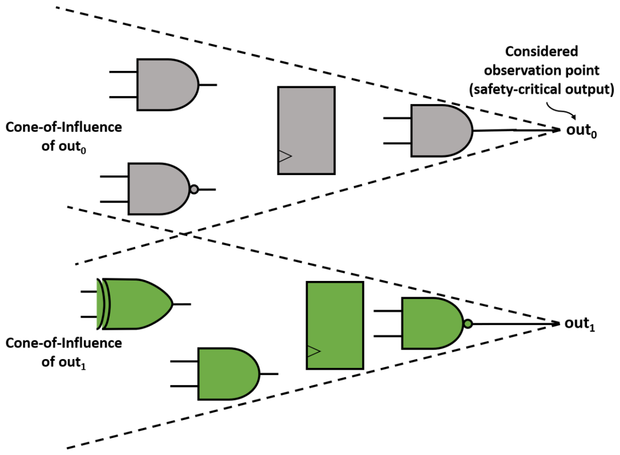

Considered observation point (safety-critical output)
out$_0$
Cone-of-Influence of out$_0$
out$_1$
Cone-of-Influence of out$_1$

Figure 2. Out-of-COI example when out_0 is the only safety critical output.

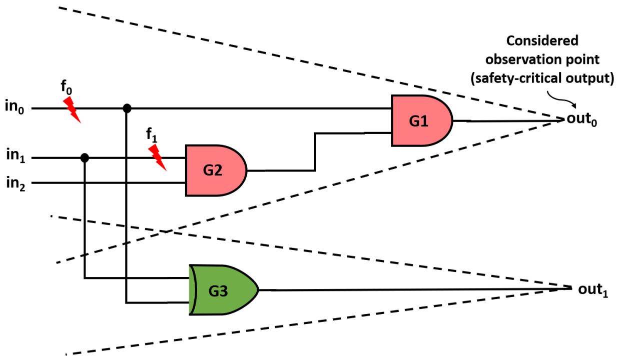


Figure 3. Unactivatable and unpropagatable analysis example.

3.3.2. Formal Analysis Check Types

As opposed to structural analysis checks for which physical connections of a design are taken into account, formal analysis checks are used to classify faults as well. The approach uses a good machine and bad machine similar to the fault simulation and injects a fault in the bad machine for formal analysis. In the end, the output signal values of good and bad machines are compared to check whether an injected fault is propagated or not. A formal tool generally generates a Boolean representation of the function implemented by the circuit (or part of it) and uses formal techniques as explained above to prove this Boolean equation. Formal analysis tools use various engines based on Boolean expressions representation and manipulation techniques, such as binary decision diagrams (BDDs) [30] to prove the formal properties exhaustively. There are two types of this analysis:

- Activation analysis: This analysis checks whether the fault can be functionally activated from the inputs. If not, then it is determined to be safe.
- Propagation analysis: This one checks whether the fault can propagate to the relevant output(s). If it cannot, then it is safe.

The technique described in Section 4 deploys both structural and formal analysis checks resorting to formal methods.

4. Proposed Application-Dependent Safe Fault Identification Method

In this section, first, we explain the definition and details of application-dependent safe faults. Then, we describe each step of the proposed technique.

In Section 3, we explain that a safe fault does not disturb any safety critical functionality because it is not located in any safety relevant logic or is in a safety relevant component. Based on this explanation, we further classify safe faults as follows:

- Structurally safe (Str-Safe): These are faults that cannot be activated or propagated to the outputs of interest by any test sequence because of the design's structural constraints. For example, a fault in the redundant logic or a floating net (i.e., any net that does not have a load) is Str-Safe. Another example is supply0 and supply1 nets. Specifically, a SA0 fault on supply0 net and a SA1 fault on a supply1 net are Str-Safe. Finally, a SA1 fault on a pull-up gate and a SA0 fault on a pull-down gate are Str-Safe.
- Functionally safe: As opposed to structurally safe faults, a test or test sequence for functionally safe faults exists, and their effects may propagate to design outputs. However, they do not affect any safety critical functionality. For example, faults in the debug unit of a CPU not used due to hardware configuration are functionally safe.

The present work focuses on a subset of functionally safe faults, corresponding to application-dependent safe faults (App-Safe). App-Safe faults are related to the SW application that the target system executes, and they cannot disturb the safety critical functionality in the operational mode. Therefore, it can be said that a fault can be App-Safe for one software application but may be dangerous for another software application.

More specifically, the target system considered in this work performs a single software application during the whole operational life. During the operation in the field, this application and its input data set do not access all the design parts; thus, inaccessible components generate App-Safe faults. For example, if the SW application does not use any multiplication operation, all resources related to the multiplication opcode become App-Safe faults. Therefore, opcodes of an SW application are a good indicator for App-Safe fault identification. Referring to the multiplication example again, when the SW application, which runs on the target design, does not include multiplication opcode, the SW application does not trigger multiplication hardware in the arithmetic logic unit (ALU), so faults on these components contribute to the App-Safe fault list. Another example of App-Safe faults can be found in the design-for-test modules of the design. The SW application does not use these hardware elements during the normal operation mode; hence, the corresponding faults are App-Safe.

In the following subsection, we explain the proposed flow to identify App-Safe faults in an industrial-size SoC when an SW application is being run on it.

4.1. The Proposed Flow

In Figure 4, the proposed flow to identify App-Safe faults is shown, step by step. At the beginning of the flow, we have a design-under-test (DUT) circuit (typically, an

SoC) and a SW application running on it. First, we run several logic simulations with different representative input data sets. The goal of running logic simulations is to analyze the design's behavior when it runs the SW application. Next, application-specific formal properties are developed to translate the design's operational behavior into the formal environment. Formal properties provide input to the formal analysis tool to identify App-Safe faults. Finally, the formal analysis tool is deployed, and safe faults are listed. In the following subsections, we discuss each step in detail.

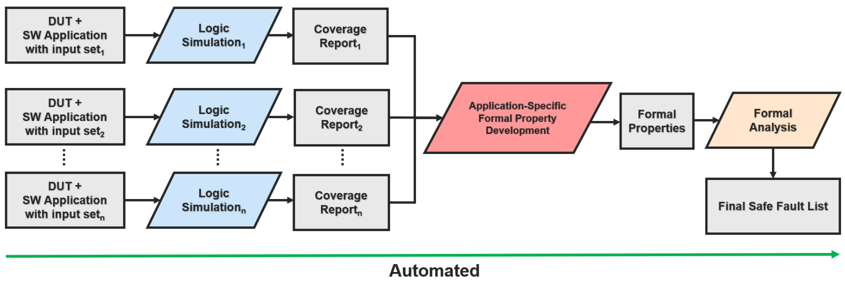


Figure 4. Proposed application-dependent safe fault identification flow.

4.1.1. Logic Simulation

In this step, we perform several logic simulations on the design under test (DUT) executing the SW application with different representative realistic data input sequences, i.e., set$_1$ to set$_n$, as shown in Figure 4. More than one logic simulation is performed when each of them runs with different input data since we aim to identify which design parts are independent of the input data set. The purpose of performing logic simulations is two-fold:

- To understand which design parts are affected by the input data set;
- To extract the design's operational behavior when it runs an SW application.

To achieve the objectives, we generate hardware design code coverage data per each logic simulation and dump them into the coverage reports.

In general, logic simulations aim to detect which points are not toggled, as these are App-Safe candidates that must be addressed. Concerning coverage metrics, the proposed work focuses on hardware code coverage that assesses how well the stimuli exercise the design code by pointing to design components that did not meet the desired coverage criteria [31]. Our technique deploys toggle and block coverage sub-types of design code coverage to identify App-Safe faults. Block coverage is a primary code coverage metric that identifies which lines in the code have been executed and which have not. On the other hand, toggle coverage monitors, collects, and reports the signal toggle activity, allowing the identification of unused signals or signals that remain at a constant value of 0 or 1.

The block and toggle coverage metrics provide insight into the SW application behavior during the operational life of an IC. Thus, we can identify App-Safe candidates included in the functionally safe fault list. More specifically, block coverage can indicate that some states are never activated, indicating that the SW application does not use the corresponding design components. Likewise, constant signals identified by toggle coverage can highlight invalid configurations, not utilized functions, among others. Moreover, the combination of block and toggle coverage data should be carefully analyzed because they can point out further information about the SW application's behavior. For the sake of an example, an untoggled signal may never activate a state machine block, and this can cause some other blocks to remain unactivated during the simulation. The small Verilog code in Listing 1 and Table 1 illustrates block coverage, toggle coverage and explains why both of them should be carefully analyzed. Listing 1 shows that rf_data_in block is never activated since $break\_error$ is never toggled to logic 1 as shown in Table 1. This coverage results also means that rf_data_in never gets the right-hand side value at line 402, as the block is not activated. This example points out the importance of assessing block and toggle coverage together.

After running logic simulations and measuring the hardware code coverage metrics presented above, the hardware coverage metrics data are available for report generation and analysis. At the end of this step, coverage reports represent the design's operational behavior under the effect of different input data sets. When this behavior is translated into formal properties, as explained in the following subsection, we call them application-specific formal properties.

Listing 1. Block coverage example: rf_data_in is not executed.

```
if(srx_pad_i | break_error)
// The following "begin" block is covered (100%)
begin
        if(break_error)
        // The following block is not covered (0%)
        rf_data_in <= {8'b0, 3'b100};
else
        // The following block is covered (100%)
        rf_data_in <= {rshift, 1'b0, rparity_error, rframing_error};
        // The following block is covered (100%)
        rf_push <= 1'b1;
        rstate <= sr_idle;
end
```

Table 1. Toggle coverage example: break_error is not toggled.

Signal Name	0-to-1 Toggling	1-to-0 Toggling
break_error	0	0

4.1.2. Application-Specific Formal Property Development

The development cycle of ICs begins with inferring the specification and requirements of the target system. Additionally, the DUT must be verified with a formulated verification plan, which is defined by both design and verification engineers. Then, features or requirements of the DUT are created and mapped to the formal properties to deploy them in a formal analysis tool [32]. Formal properties are created from the design specification and implementation decisions. Thus, after extracting the target system's operational behavior through logic simulations, in this step, we translate this behavior to the formal properties to be used in a formal analysis tool, which will identify additional App-safe faults.

We use two types of formal properties to define the correct behavior of the design. The first one is assume statement, which creates an assumption for the specified Boolean expression that evaluates to either true or false. In the general sense, it specifies that the given property is an assumption and is used to generate the input stimulus. Hence, assume statements can be helpful when we define a design configuration or to inform the tool how the design inputs can behave. Without this assumption, a formal tool checks all possible input combinations of the DUT. There are two benefits of using assume statements in the formal environment. First, it allows excluding illegal input combinations when known. Legal inputs are those that we expect to see during normal operation. It is not realistic to expect the design to behave correctly when all possible input combinations are being applied, unless we explicitly define every possible set of input combinations that the design can theoretically see. The second benefit of using assume statements is that it intentionally reduces the state space, which is exhaustive when no assumption is defined. For example, as we want to prepare our formal environment considering the design's operational behavior, we should disable the *scan_enable* pin, as the scan chain is not activated during the operation and is used only for test purposes. In this case, the assume statement given in (2) is created to inform the formal tool about the *scan_enable* signal behavior; thus, the input test stimuli of a formal analysis tool are limited accordingly. The command given in (2) simply informs the tool that *scan_enable* is always logic-0. Assume statements also increase the safe fault identification capacity of a formal analysis tool by guiding it. Moreover, similar to the example given below, the input ports of design instances are suitable candidates for assume statements.

$$assume - env \{scan\_enable == 1'b0\} \tag{2}$$

The second formal property is the fault propagation barrier, which creates a formal barrier that blocks the propagation of a fault. In this case, faults cannot propagate after this barrier; therefore, they cannot disturb any safety critical functionality. For instance, knowing that the debug unit is not used in the design's operational mode, we can block all faults to propagate from it and identify more App-Safe faults. As seen in (3), the formal analysis tool is asked to block all faults propagated to *du_dat_o*, which is the debug unit's data output signal. As in this given example, output ports are proper candidates for a fault propagation barrier as opposed to assume statements, for which input ports are suitable candidates.

$$check\_fsv - barrier \{du\_dat\_o\} \tag{3}$$

Consequently, the application-specific formal properties [33] can be developed using assume statements and fault propagation barriers. By doing so, the internal architecture and logical details of the target system, the operational constraints (if any), or the initial configuration of the design can be defined as formal properties to be used in the formal analysis step. Therefore, the design's operational behavior can be transferred from the logic simulations into the formal analysis tool. The following subsection explains how the formal analysis tool uses these application-specific formal properties.

4.1.3. Formal Analysis

Having specified the formal properties of a target design in a suitable notation, a formal analysis tool can be employed to generate App-Safe faults. The advantage of the formal analysis is that it provides a precise answer to whether a fault is propagated since it considers all possible input stimuli combinations (yet configured and limited thanks to *assume statements* as explained before) and hence, it eliminates the dependency on input stimuli. In this step of our flow, a formal analysis tool checks each fault in the target design to see whether it can be propagated to the observation points or not. If any input stimuli cannot propagate a fault, it is classified as safe; in our case, it is App-Safe. Otherwise, the fault falls into the dangerous category.

The formal analysis flow, which includes three phases, is shown in Figure 5. Phase I begins with the creation of input files that are the formal properties established in the previous stage and the DUT. Then, it continues with the development of the tool command language (TCL) setup script for the formal analysis tool. The setup script consists of Verilog files, libraries, and formal property files. The setup script first analyzes design and property files to check for syntax errors. Then, it defines clock and reset signals. The clock definition is to specify the characteristics of how the clock is driven during a formal analysis run. The reset specification aims at bringing the design to a known state and avoiding unreachable failure states. In the next step, warnings are generated by the formal analysis tool if there is a mismatch between formal properties and the DUT. For example, a signal tied to the ground in the DUT and the assume statement that defines this signal as if it is always logic-1 can create a mismatch, and a warning is generated. However, as we automatically translate coverage reports to the formal properties, this is not the case for the work proposed in this work. Then, in Phase II, the formal engine proves the formal properties by running the structural and formal checks as presented in Section 3.3. Finally, in Phase III, App-Safe faults are identified and reported.

In brief, a formal analysis tool uses formal properties to generate safe faults. When we include formal properties driven by SW application, as mentioned before, we enable the tool to work in a well-specified configuration. Hence, formal analysis with the formal properties increases the number of identified App-Safe faults.

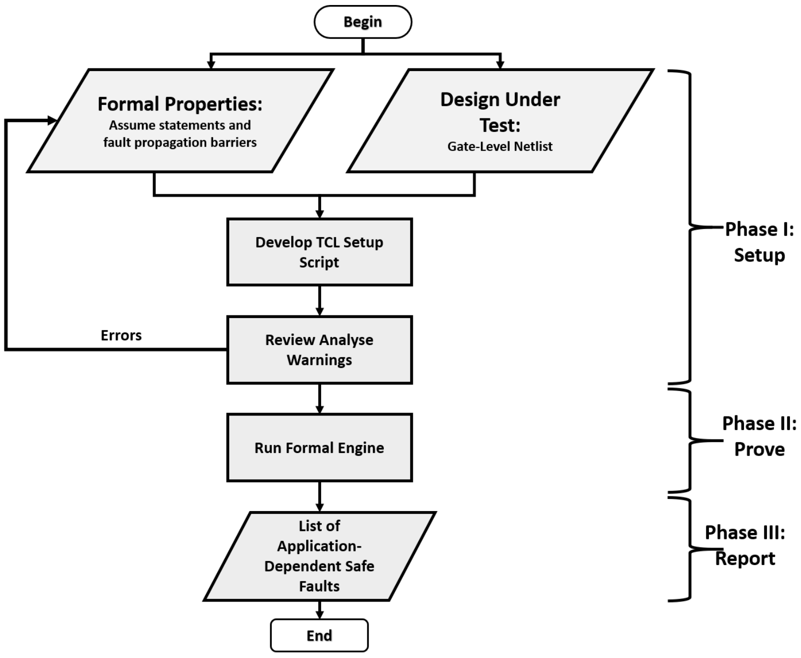


Figure 5. Formal analysis phases.

5. Case Study: The AutoSoC Benchmark Suite

The proposed application-dependent safe fault identification method is evaluated on the AutoSoC benchmark suite, which we conceptualized in [8]. The AutoSoC is an open-source benchmark suite, incorporating all required elements in the format of a configurable SoC. It is developed to support research in the automotive domain by providing varied hardware configurations, safety mechanisms, and representative software applications. In this section, we explain the AutoSoC by detailing its CPU and other functional blocks.

5.1. General Architecture of the AutoSoC

Developed by characterization of commercial CPUs used in the automotive field, the AutoSoC has two main processing units as the safety island and the application specific block, as illustrated in Figure 6. While the safety island handles all safety-critical processes driven by ISO 26262 [4], the application-specific block executes the hardware needed for application-specific processing. It is also important to note that the safety island and the application-specific block have dedicated software stacks to execute distinguished applications. The interconnect block deploys Wishbone Bus for internal SoC communication. Additionally, the remaining blocks in Figure 6 are included to fulfill the requirements for communication, security, and general infrastructure.

Since the AutoSoC is implemented as a modular, it has several configurations as detailed in [8]. One of these configurations named *AutoSoC* QM is used in this work. This configuration is a fully functional version of the benchmark suite. When considering functional blocks of the AutoSoC shown in Figure 6, the *AutoSoC QM* configuration has only the application-specific block, but here, the presented work remains suitable to all available configurations of the AutoSoC.

The main CPU used in the development of the AutoSoC is the *mor1kx* implementation of the OpenRISC [34]. This implementation provides all necessary tools and examples for developing SoCs, such as CPU, memory, debug unit, communication protocols, and a bus. Concerning software resources, the AutoSoC includes several options, some of which come from the *mor1kx* package and the others developed by ourselves in conformity with automotive functional safety analysis. These software resources are available as both BareMetal, and the real-time executive for multiprocessor systems (RTEMS) real-time operating system [35]. Furthermore, the automotive cruise control application (CCA) is developed and targeted for safe fault identification. This application is based on BareMetal and the RTEMS operational system and covers several tasks: reading vehicle sensor data from UART and CAN, computing actuation, and setting engine parameters. In addition, the AutoSoC has STL programs that target the CPU (mor1kx_cpu). These STL programs are developed for online testing of the AutoSoC. The current available STL presented in the open-source AutoSoC package comprises 16 test programs [8].

Finally, the AutoSoC is available at both the register-transfer level (RT-Level) and gate level. The synthesis is performed using Cadence GPDK045 (45nm CMOS Generic Process Design Kits). The proposed approach in this paper is demonstrated using the gate-level model of the AutoSoC.

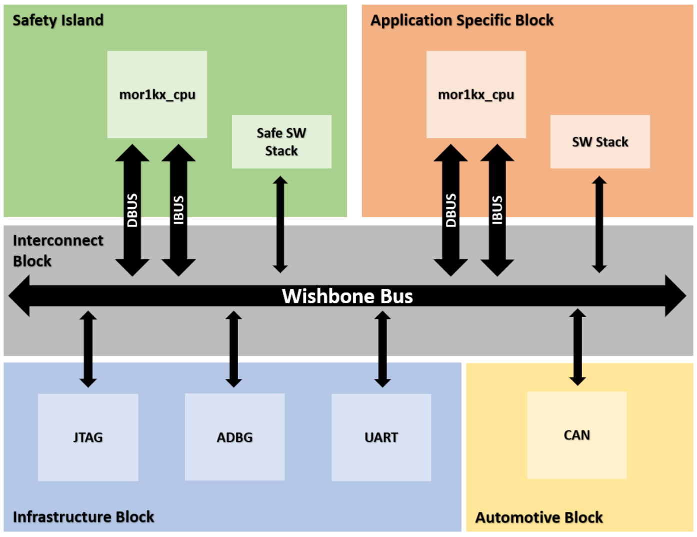


Figure 6. AutoSoC functional blocks.

5.2. UART IP

The AutoSoC benchmark suite includes a UART IP, which incorporates the industry standard National Semiconductors' 16550A device features. Furthermore, as it is a well-known and widely-used communication standard by the industry and academia, and the proposed safe fault identification method is also extended to the UART. In this subsection, we provide details about the adopted core.

UART is a block of circuitry that uses asynchronous serial communication with configurable speed. It operates data transfer by receiving data from a peripheral device or a CPU. Moreover, the UART includes an interrupt system and control capability tailored to minimize software management of the communication link. The UART IP used in the AutoSoC operates in a 32-bit bus mode fully compatible with Wishbone Bus. As depicted in Figure 7, the UART core consists of receive logic, control, and status registers, modem control module, transmit logic, Baud generator logic, and interrupt logic. Incoming serial messages are received by the RX shift register, whose Baud rate is programmable through Baud generator logic. Received messages are placed in the receive FIFO if the incoming messages have no problems. On the contrary, the TX shift register handles the transmission of data written to the transmit FIFO. Control and status registers allow the specification and observation of the format of the asynchronous data communication used. Modem control has registers that allow transferring control signals to a modem connected to the UART. The UART IP also has Baud generator logic to control transmit and receive data rates. Finally, interrupt logic allows enabling and disabling interrupt generation by the UART.

The AutoSoC benchmark suite includes the above-explained UART IP and some test programs to experiment with the functionality of the UART to provide a baseline for researchers to develop and validate their approaches.

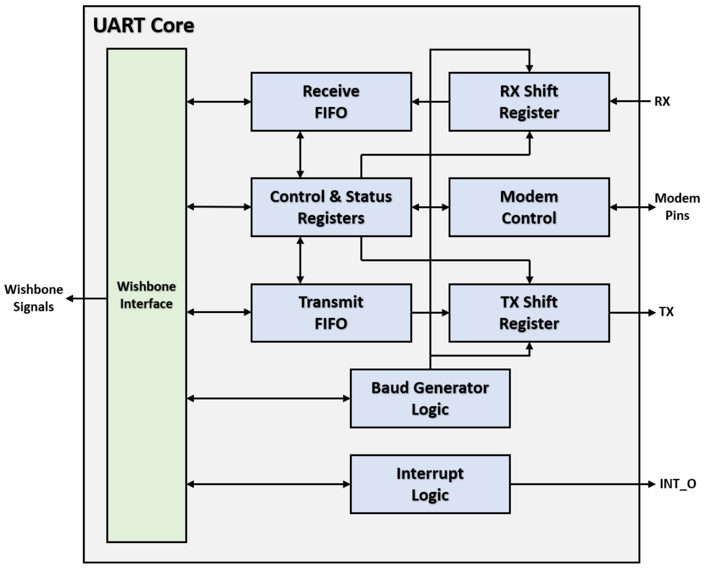


Figure 7. Block diagram of the UART IP.

5.3. CAN Controller IP

The CAN is a communication bus standard introduced by Bosch in 1986. It is intended to work in the automotive field for serial communication applications among microcontroller units. The CAN has several benefits; it is low-cost, and it has the ability to self-diagnose and repair data errors. These features promote CAN's popularity in the automotive and some other industries, such as medical or aerospace [36]. As it represents the automotive industry's challenges, we validate the proposed safe fault identification method on CAN.

The AutoSoC benchmark suite has open hardware implementation of the SJA1000 [9], which is a standalone controller for the CAN, developed by Philips Semiconductors in the early 2000s. Figure 8 shows the block diagram of SJA1000 CAN. The CAN transceiver is a module to connect other nodes to the CAN. The CAN core block controls the reception and transmission of CAN frames. The interface management logic implements the CAN interface as a link to the host CPU through its set of registers. Additionally, this block configures the operational mode of CAN, whether it works in *BasiCAN* or *PeliCAN* mode. The transmit buffer stores messages in extended or standard format. The CAN core block reads messages from the transmit buffer whenever the interface management logic forces it. The acceptance filter comes into prominence when receiving a message. It checks whether the message on the bus has to be stored by the CAN or not. All received messages accepted by the acceptance filter are stored in the receive FIFO.

As the AutoSoC benchmark suite uses a Wishbone Bus, the adopted CAN is directly connected without the need for bridges between different bus interfaces. When it is required to add another node to be communicated with the Host CPU, the CAN Transceiver provides a straightforward way for connection. Moreover, the AutoSoC benchmark suite provides an STL for the self-test of the CAN. As it is explained in [37], the developed STLs implement an effective in-field test for the CAN based on a functional approach and also provide experimental evidence to demonstrate its effectiveness.

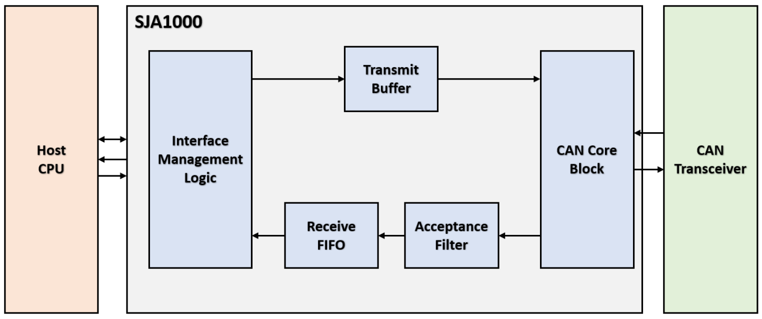


Figure 8. Block diagram of the adopted SJA1000 CAN controller IP.

6. Experimental Setup and Results

This section first describes the experimental setup we used to quantitatively assess the effectiveness of the proposed approach. Then we provide the results in a separate subsection by focusing on CPU, UART, CAN, and finally the combined results.

6.1. Experimental Setup

In order to demonstrate the effectiveness of the proposed App-Safe fault identification method, we used the experimental setup shown in Figure 9. Our setup is composed of two AutoSoC nodes; each includes a CAN and a UART IP to communicate with each other, and one of the two AutoSoCs (named AutoSoC-0) is assumed to be active, whereas the other (named AutoSoC-1) is the passive node. Moreover, CCA accesses CAN or UART in both the AutoSoC-0 and the AutoSoC-1. Thus, each CCA comes in two modes, even though the executed steps are symmetric; the two AutoSoC nodes alternatively receive and send messages in the same configuration. Furthermore, even if it is changeable, AutoSoC-0 receives messages first, while AutoSoC-1 transmits first in our experimental setup. Finally, the whole system is simulated at the gate level.

Concerning the EDA tools, we used Cadence Xcelium™ for logic simulations, Cadence® Integrated Metrics Center (IMC) for coverage analysis, Cadence® JasperGold® Functional Safety Verification (FSV) App for formal analysis, and Cadence® Xcelium™ Fault Simulator (XFS) for the fault simulation. However, the approach proposed in this paper remains applicable to other tool flows as well.

In brief, we first performed logic simulations using the hardware configuration described before, which runs the CCA SW application using different input data sets, as shown in Figure 4. Then, coverage reports are generated and translated into application-specific formal properties that configure the formal analysis environment according to the SW application's behavior. Finally, the formal analysis tool is deployed to identify App-Safe faults.

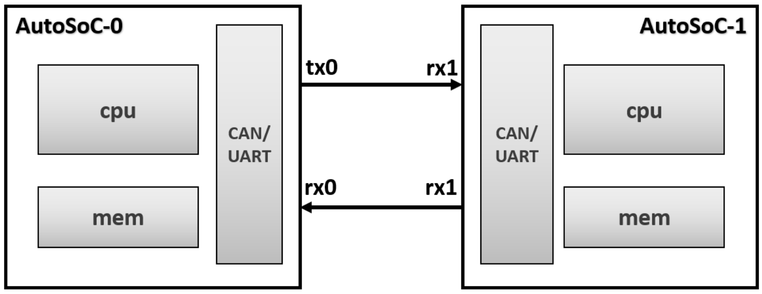


Figure 9. Experimental setup composed of two AutoSoC nodes.

6.2. Experimental Results

This subsection presents the identified safe faults in CPU, UART, and CAN, respectively. Then, the combined results (fault simulation + formal analysis) are reported for the CPU and CAN modules (this step does not include the analysis of UART).

6.2.1. Safe Faults in CPU

Firstly, App-Safe fault identification is checked in the CPU core (which has 96,354 faults in total) when it runs a SW application. We summarize the safe fault results of the CPU in Table 2.

Table 2 categorizes the results based on the analysis we run. In the top row, it can be seen that we performed four analyses as follows:

- Application-independent: The formal analysis tool is deployed on the gate-level netlist of the AutoSoC without any formal properties, meaning that the identified safe faults are valid for any SW application.
- BareMetal-CCA: The CCA runs BareMetal, which refers to running the SW application directly on a CPU without the support of an operating system. In order to perform this analysis, the gate-level netlist of AutoSoC and the formal properties (as explained in Section 4.1.2) are used as inputs to the formal analysis tool.
- RTEMS-CCA: Unlike BareMetal-CCA, the SW application runs on an operating system in this analysis, meaning that it can start and stop different processes concurrently. The RTEMS-CCA causes higher signal activity when compared to BareMetal-CCA, as it runs on operating systems that trigger more signals. In addition, RTEMS-CCA uses two additional opcodes compared to BareMetal-CCA. This means that RTEMS-CCA triggers more design components than BareMetal-CCA.
- BareMetal-Sum: For this analysis, we use an entirely different SW application than CCA. The application performs a sum operation, and it has fewer opcodes than BareMetal-CCA. This SW application aims to show how App-Safe faults change when the CPU is running a different application.

In brief, App-Safe faults are originated from what a SW application executes in an IC. For example, some design components are not accessed during the design's operational life, such as debug units or scan chains. In addition, unused opcodes cause App-Safe faults, meaning that if (for example) the multiplication opcode is not used in the SW application that runs on the IC, all signals related to multiplication hardware become App-Safe faults, as they are not exercised. Table 2 reports the results for the CPU core, also detailing the results achieved on each component module inside it. In the application-independent analysis, the formal analysis tool identifies 8.785% safe faults with respect to all faults in the CPU. We highlight that all the identified safe faults in the application-independent analysis are Str-Safe faults because the formal tool could not identify any safe faults using the formal fault analysis check types mentioned in Section 3.3.2 without formal properties.

Concerning the three application-dependent analyses (BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum):

- The *top* module includes connectivity signals and configuration-related signals. Among these, debug unit's address and data signals, interrupt request signals, multicore configuration signals, special-purpose-register signals are identified as safe in all analyses since they are not activated due to the SW applications configuration. Depending on the opcodes used in the applications, there are slight differences in BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum. For example, RTEMS-CCA triggers exception signals, which are connected to the top level.
- The *Decode_execute Unit* is the module where the instruction memory management unit (IMMU) and the data memory management unit (DMMU) signals take part. Many safe faults are identified in the IMMU and DMMU, which are not used by the SW applications. The number of safe faults is different between BareMetal-CCA and RTEMS-CCA because of the exception signals used by RTEMS-CCA, as mentioned above. In addition, the deviation between BareMetal-CCA and BareMetal-Sum is due to division and multiplication-related signals, which BareMetal-Sum does not use.
- The *load–store unit* computes the addresses used by load and store instructions. Safe faults may exist, as not all addresses are used by the SW applications. In addition, some connection signals create a slight difference between BareMetal-CCA and RTEMS-CCA.
- The *fetch stage* fetches the next instruction from memory into the instruction register. Therefore, it is directly associated with the address range, which is not fully covered by the SW application. Therefore, safe faults can be identified in this unit. In addition, the difference between BareMetal-CCA and RTEMS-CCA is due to exception signals.
- The *control stage* has the most considerable impact on the number of identified safe faults. This unit contains features such as a tick timer, interrupts, and configuration registers. Since the CPU configuration is the same in all applications, configuration registers create the same amount of safe faults. However, the tick-timer unit has a higher activity in RTEMS-CCA; hence, it has fewer safe faults when the CPU runs RTEMS-CCA.
- Concerning the *arithmetic logic unit*, the proposed technique identifies the same amount of safe faults in BareMetal-CCA and RTEMS-CCA, as they use the same arithmetic opcodes. However, BareMetal-Sum performs only addition operations; therefore, all the other arithmetic operations contribute to the safe faults.
- The *decode unit* is directly affected by the used opcodes; hence, there is a difference between the numbers of safe faults, as all three analyses use different numbers of opcodes.

The results in Table 2 show that the percentage of safe faults varies widely from one module to another, depending on the tasks performed by the modules. In addition, the number of App-Safe faults is relevant, accounting for about 20%, 14%, and 40% in BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum applications, respectively.

Table 2. Safe faults in CPU.

CPU Modules	Application-Independent		Baremetal-CCA		RTEMS-CCA		Baremetal-Sum	
	Safe Faults	**Safe Faults w.r.t. Total Faults**	**Safe Faults**	**Safe Faults w.r.t. Total Faults**	**Safe Faults**	**Safe Faults w.r.t. Total Faults**	**Safe Faults**	**Safe Faults w.r.t. Total Faults**
Top	1679	1.743%	1725	1.790%	1716	1.781%	1717	1.782%
Register File	2	0.002%	5	0.005%	2	0.002%	5	0.005%
Decode_Execute Unit	651	0.676%	844	0.876%	719	0.746%	949	0.985%
Load Store Unit	910	0.944%	2380	2.470%	2317	2.405%	2380	2.470%
WriteBack Mux Unit	0	0.000%	0	0.000%	0	0.000%	76	0.079%
Fetch Stage	976	1.013%	1230	1.277%	1195	1.240%	1230	1.277%
Control Stage	3966	4.116%	11,618	12.058%	6418	6.661%	11,618	12.058%
Arithmetic Logic Unit	55	0.057%	1000	1.038%	1000	1.038%	19,478	20.215%
Decode Unit	5	0.005%	267	0.277%	16	0.017%	315	0.327%
Branch Prediction Unit	0	0.000%	0	0.000%	0	0.000%	0	0.000%
TOTAL	8465	8.785%	19,484	20.221%	13,670	14.187%	38,193	39.638%

6.2.2. Safe Faults in UART

Concerning the UART module, which has 19,120 faults in total, we followed the same procedure using two scenarios (application-independent, and CCA is compared), and the results are detailed in Table 3. We also noted that there is no difference between BareMetal-CCA or RTEMS-CCA, so we only report the identified safe faults as CCA in Table 3. In short, the proposed technique identified 11.088% safe faults, which is two times more than when compared to the application-independent analysis.

More specifically, we have the following:

- The *regs* unit has configuration registers, whose value is written in the initialization phase. Since the UART configuration is fixed in CCA, some parts of the UART are unused; thus, several safe faults can be identified in this unit.
- Safe faults in the *transmitter* module originate from the configuration of the transmission format, such as the selected BAUD rate. Therefore, more safe faults can be found in this unit when the SW application is fixed, as in this work. Correspondingly, *transmitter fifo* is partially affected by these factors.
- Concerning the *receiver* module that is directly affected by the configuration registers, a significant amount of increase in the number of safe faults is observed. This mainly stems from the fact that the receiver module is responsible for generating interrupts. However, the CCA works in polling mode, meaning that no interrupt is used. Moreover, the receiver module has a modem configuration, which CCA does not need. By extension, *receiver fifo* is partly affected, similar to transmitter fifo.

Table 3. Safe faults in the UART IP.

UART Modules	Application-Independent		CCA	
	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults
Top	9	0.047%	19	0.099%
wb_interface	78	0.408%	78	0.408%
regs	357	1.867%	1003	5.246%
transmitter	67	0.350%	67	0.350%
uart_sync_flops	6	0.031%	6	0.031%
fifo_tx	101	0.528%	101	0.528%
receiver	171	0.894%	651	3.405%
fifo_rx	195	1.020%	195	1.020%
TOTAL	984	5.146%	2120	11.088%

6.2.3. Safe Faults in CAN

The same analysis is performed for the CAN module, which has 38,012 faults in total, and the results are provided in Table 4. In the application-independent analysis, the formal analysis tool can classify only 1.415% of all faults as safe. On the other hand, when the proposed approach is deployed, the amount of safe faults is increased to 12.909%, which is not negligible.

Similar to UART, the number of safe faults in CAN is directly affected by its configuration. In CCA, we configure the CAN to work in peliCAN mode, which has extended frame format messages. When the basiCAN mode is used, more safe faults can be identified. To put the results given in Table 4 more explicitly, we have the following:

- *Acceptance_code_mask* defines whether the corresponding incoming bit is compared to the respective bit in the *acceptance_code_regs*. Similarly, *bus_timing_regs* defines the values of the Baud rate prescaler and programs the period of the CAN system. Moreover, *clock_divider_regs* controls the clock frequency for the microcontroller and allows to deactivate the clock pin. In addition, the CCA works in polling mode, so

safe faults can be found in the *IRQ* registers. Consequently, all these registers should not be changed after the initial configuration; thus, this creates additional safe faults.
- *Bit timing logic* is directly affected by *bus_timing_regs* explained above, so the CCA originates some safe faults in this unit.
- *Bit stream processor* corresponds to the control and processing unit of the peripheral. It is a sequencer that controls the data stream between the transmit buffer, the receive fifo, and the CAN bus. Additionally, error-detection, arbitration, stuffing, and error-handling are done in this unit. In addition, the bit stream processor is affected by the configuration, such as working mode of the CAN, such as the listen-only mode or self-test mode. The CCA does not use these modes, which provide the safe faults shown in Table 4.
- *Acceptance filter* checks whether the message currently on the bus has to be stored by the peripheral or not. If the message is accepted, it is stored in the fifo. In other words, the bit acceptance filter and its fifo are related to *acceptance_code_regs* and *acceptance_code_mask*; therefore, the fixed content of these registers gives rise to safe faults.

Table 4. Safe faults in the CAN controller IP.

CAN Modules	Application-Independent		CCA	
	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults
Top	10	0.026%	41	0.108%
can_registers	22	0.058%	769	2.023%
acceptance_code_regs	0	0.000%	52	0.137%
acceptance_mask_regs	0	0.000%	52	0.137%
bus_timing_regs	0	0.000%	26	0.068%
clock_divider_regs	11	0.029%	41	0.108%
command_reg	13	0.034%	57	0.150%
error_warning_reg	10	0.026%	74	0.195%
irq_en_reg	0	0.000%	15	0.039%
mode_regs	14	0.037%	53	0.139%
tx_data_regs	0	0.000%	115	0.303%
Bit Timing Logic	46	0.121%	299	0.787%
Bit Stream Processor	354	0.931%	2988	7.861%
can_crc_rx	0	0.000%	0	0.000%
Acceptance Filter	3	0.008%	256	0.673%
can_fifo	55	0.145%	69	0.182%
TOTAL	538	1.415%	4907	12.909%

6.2.4. Combined Results: Fault Simulation and Formal Analysis

In this step, we combine the fault simulation and formal analysis, as it is proposed in this work, to check the increase in the DC. This analysis targets the CPU and the CAN modules in the AutoSoC.

As mentioned in Section 3.2, the fault simulation is not enough to classify all faults because workloads used for fault simulation cannot activate and propagate all faults. Therefore, some faults become undetected as a result of fault simulation. It is needed to analyze these undetected faults to check if the desired DC is reached. If the DC does not match the requirements, then the undetected faults must be re-analyzed using alternative methods, such as the proposed technique in this work. In short, the purpose of this step is to show that the proposed technique can increase DC to achieve the figures required by a given automotive safety integrity level.

In order to perform this analysis, we resorted to the software-based self-test (SBST) [5] approach in the form of STLs. In the considered scenario, the AutoSoC runs BareMetal-CCA in the field, and the STL, when activated, forces the processor to execute a proper sequence of instructions. Then, a signature is produced based on the generated results, and the application can compare it with the expected results if there are faults.

The developed STL for the AutoSoC CPU is a combination of 57 test programs, partly taken from [8] and partly newly developed for this paper. Concerning the STL for CAN, we use the same test programs described in [28]. The STL was developed as a collection of tasks that can either operate independently or collectively, depending on the self-test time slot [37].

The following steps are applied:

- First, Str-Safe faults are identified using the Cadence® JasperGold® Functional Safety Verification (FSV) App.
- Second, we use the Cadence® Xcelium™ Fault Simulator to inject SA0 and SA1 faults at cell ports of the AutoSoC CPU and CAN modules, which run the STL as a workload. As a result, faults are classified as detected or undetected.
- Third, DC is calculated by using (1).
- Fourth, App-Safe faults are identified before being excluded from undetected faults. This process is incremental, always focusing on faults that were previously undetected.
- Finally, DC is calculated again with the newly achieved numbers using (1).

Figures 10 and 11 detail the results of the STL efficiency and uptrend in DC when App-Safe faults are identified. Concerning the analysis in CPU, Figure 10 shows that 8465 Str-Safe faults are identified in the beginning. Then, when fault simulation is deployed, 71,255 detected and 16,634 undetected faults are classified. After fault simulation, DC is 81.07%, calculated using (1). Then, by applying the proposed safe fault identification technique using formal methods, 5627 App-Safe faults are identified, i.e., undetected faults are reduced to 11,007. Using again (1), DC is increased to 86.62%. A similar analysis is performed in CAN as shown in Figure 11. As a result, DC is increased from 88.04% to 91.97%.

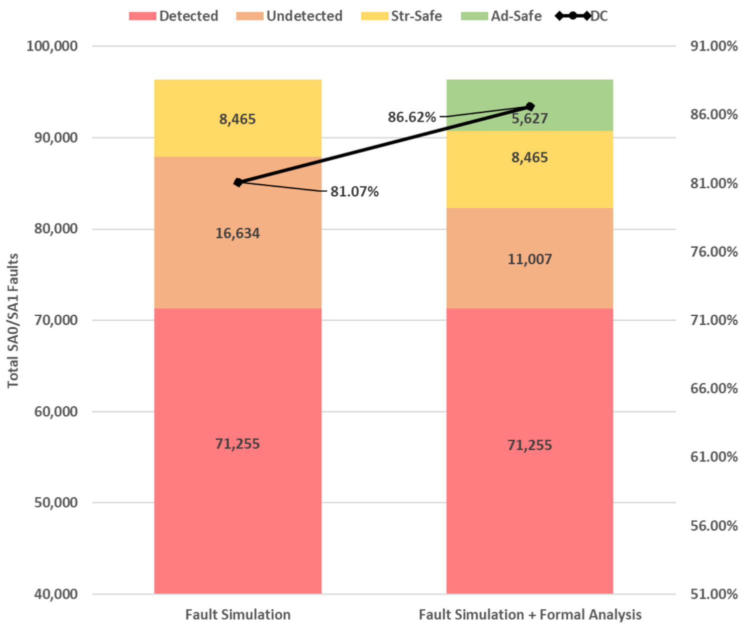


Figure 10. Combined results in CPU: uptrend in DC when fault simulation and formal analysis combined.

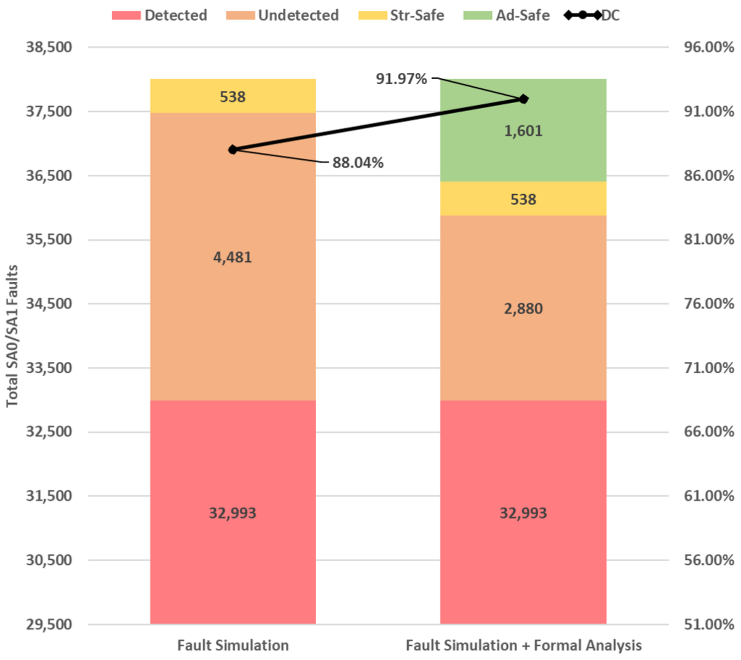


Figure 11. Combined results in CAN: uptrend in DC when fault simulation and formal analysis combined.

The proposed technique appears to be a promising way for the classification of undetected faults via safe fault identification. The combined results show that DC is improved by around 6% for the CPU and 4% for the CAN. Moreover, with a final DC of 91.97%, the CAN achieves the requirements for an automotive ASIL B [4] hardware component as is, i.e., without design modifications.

6.3. Discussion

Safety standards (e.g., ISO 26262) mandate the estimation of the achieved safety level, which in turn requires the identification of safe faults. This work provides a new technique for automatically identifying safe faults in the CPU and peripherals. The proposed technique can significantly reduce the cost and effort for safe fault identification, showing that the method can identify a significant number of safe faults. Safety standards (e.g., ISO 26262) mandate the estimation of the achieved safety level, which in turn requires the identification of safe faults. This work provides a new technique for automatically identifying safe faults in the CPU and peripherals. The proposed technique can significantly reduce the cost and effort for safe fault identification, showing that the method can identify a notable number of safe faults.

The main advantage of the proposed method is its automated approach for safe fault identification using the automotive representative hardware and software application. The method reduces the constraints of manual expert-based analysis, so the time and complexity of verification efforts are reduced simultaneously. This also helps reduce the time-to-market criteria, which is one of the biggest challenges of the IC design industry. Moreover, the proposed method is systematic and established based on logic simulation and formal analysis, supported by industrial-grade tools that make it suitable for the automotive industry and research on functional safety verification. It enables an accurate safety metrics evaluation; therefore, it allows compliance with ISO 26262 functional safety metrics. Concerning disadvantages, there is no analytical calculation regarding the number of logic simulations to be run. We ran several logic simulations using different but realistic input data sequences in our work (as explained in Section 4.1.1) and stopped running new ones when the coverage reports were the same. Additionally, concerning the computational complexity of the proposed technique, it depends upon the number of faults that are being evaluated by the formal analysis.

7. Conclusions

Functional safety verification is a crucial and non-negotiable requirement that must be considered throughout the safety critical IC design cycle. Therefore, the ISO 26262 functional safety standard was developed to guide how this requirement is implemented. According to ISO 26262, random hardware failures can occur unpredictably during the lifecycle of an IC. Thus, random hardware faults must be classified based on their effects, i.e., on whether they can disrupt any safety critical functionality or not. Nevertheless, this classification process is expensive and error prone since it requires a combination of tools and inputs from experts based on their design knowledge. The method proposed in this work brings a solution to this challenge.

The proposed methodology focuses on identifying safe faults on a SoC when it runs a single SW application. We extend functionally safe faults by the identification of application-dependent safe faults. The flow relies on code coverage analysis through logic simulations and formal methods. The methodology starts with the analysis of code coverage to understand the target system's operational behavior. In other words, faults that do not disturb any safety critical functionality are first identified through code coverage analysis. Then, code coverage results are translated to formal properties, then transferred to a formal analysis tool to constrain the environment to identify safe faults. The proposed methodology is demonstrated on the AutoSoC using its CPU, UART, and CAN when the cruise-control application runs.

We computed the number of identified safe faults (specifically focusing on stuck-at faults). In addition, we combined fault simulation and the proposed formal technique to show the increase in diagnostic coverage. As a result, the number of safe faults accounts for 20%, 11%, and 13% in the CPU, CAN, and UART modules, respectively. Concerning the diagnostic coverage, we show that it is increased by 6% and 4% in CPU and CAN modules, respectively. This analysis also proves that the number of undetected faults for the same STL is reduced by 1.5–1.6 times for the CPU and CAN, significantly increasing the diagnostic coverage for an industry-scaled SoC with a sample automotive application.

In future work, we plan to analyze different automotive representative software applications with various scenarios to see the effect of the proposed method on the safety metrics. Furthermore, an FMEDA will be created, and safety metrics guided by ISO 26262 will be calculated when the proposed technique identifies safe faults. Additionally, different STLs that target the AutoSoC and peripherals will be deployed, and the increase in the safety level will be analyzed.

Author Contributions: Conceptualization, A.C.B. and F.A.d.S.; data curation, A.C.B.; formal analysis, A.C.B.; investigation, A.C.B. and F.A.d.S.; methodology, A.C.B. and F.A.d.S.; software, A.C.B.; writing—original draft preparation, A.C.B.; supervision, M.J., M.S.R., S.H. and C.S.; writing—review and editing, A.C.B., F.A.d.S., M.S.R., S.H., M.J. and C.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by project RESCUE funded from the European Union's Horizon 2020 research and innovation program under the Marie Sklodowaska-Curie grant agreement No. 722325.

Data Availability Statement: The AutoSoC Benchmark Suite is open source and available at https://www.autosoc.org (accessed on 18 December 2021) Additionally, the original contributions presented in the study are included in the article, and further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ADAS	Advanced Driver Assistance System
ALU	Arithmetic Logic Unit
ASIL	Automotive Safety Integrity Level
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BIST	Built-in Self-Test
CAN	Controller Area Network
CCA	Cruise Control Application
CPU	Central Processing Unit
COI	Cone-of-Influence
DC	Diagnostic Coverage
DUT	Design Under Test
ECU	Electronic Control Unit
FSV	Functional Safety Verification
IC	Integrated Circuit
IMC	Integrated Metrics Center
RTEMS	Real-Time Executive for Multiprocessor Systems
RTL	Register Transfer Level
SBST	Software-Based Self-Test
SoC	System-on-Chip
STL	Software Test Library
SW	Software
TCL	Tool Command Language
UART	Universal Asynchronous Receiver–Transmitter
XFS	Xcelium Fault Simulator

References

1. Jenihhin, M.; Sonza Reorda, M.; Balakrishnan, A.; Alexandrescu, D. Challenges of Reliability Assessment and Enhancement in Autonomous Systems. In Proceedings of the 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Noordwijk, The Netherlands, 2–4 October 2019; pp. 1–6. [CrossRef]
2. Munir, A. Safety Assessment and Design of Dependable Cybercars: For today and the future. *IEEE Consum. Electron. Mag.* **2017**, *6*, 69–77. [CrossRef]
3. Nardi, A.; Armato, A. Functional safety methodologies for automotive applications. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; pp. 970–975. [CrossRef]
4. International Standardization Organization. *ISO 26262 Road Vehicles—Functional Safety*, 2nd ed.; ISO: Geneva, Switzerland, 2018.
5. Psarakis, M.; Gizopoulos, D.; Sanchez, E.; Sonza Reorda, M. Microprocessor Software-Based Self-Testing. *IEEE Des. Test Comput.* **2010**, *27*, 4–19. [CrossRef]
6. Cantoro, R.; Firrincieli, A.; Piumatti, D.; Restifo, M.; Sanchez, E.; Sonza Reorda, M. About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications. In Proceedings of the 2018 IEEE 19th Latin-American Test Symposium (LATS), São Paulo, Brazil, 12–16 March 2018; pp. 1–6. [CrossRef]
7. Benso, A.; Di Carlo, S. The Art of Fault Injection. *Control Eng. Appl. Inform.* **2011**, *13*, 9–18.
8. da Silva, F.A.; Cagri Bagbaba, A.; Ruospo, A.; Mariani, R.; Kanawati, G.; Sanchez, E.; Sonza Reorda, M.; Jenihhin, M.; Hamdioui, S.; Sauer, C. Special Session: AutoSoC—A Suite of Open-Source Automotive SoC Benchmarks. In Proceedings of the 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 5–8 April 2020; pp. 1–9. [CrossRef]
9. SJA1000, Stand-Alone CAN Controller. Available online: https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf (accessed on 18 December 2021).
10. Alexandrescu, D.; Evans, A.; Glorieux, M.; Nofal, I. EDA support for functional safety—How static and dynamic failure analysis can improve productivity in the assessment of functional safety. In Proceedings of the 2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS), Thessaloniki, Greece, 3–5 July 2017; pp. 145–150. [CrossRef]
11. Lu, K.L.; Chen, Y.Y.; Huang, L.R. FMEDA-Based Fault Injection and Data Analysis in Compliance with ISO-26262. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg, 25–28 June 2018; pp. 275–278. [CrossRef]
12. Juez, G.; Amparan, E.; Lattarulo, R.; Rastelli, J.P.; Ruiz, A.; Espinoza, H. Safety assessment of automated vehicle functions by simulation-based fault injection. In Proceedings of the 2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES), Vienna, Austria, 27–28 June 2017; pp. 214–219. [CrossRef]

13. Fu, Y.; Terechko, A.; Bijlsma, T.; Cuijpers, P.J.L.; Redegeld, J.; Örs, A.O. A Retargetable Fault Injection Framework for Safety Validation of Autonomous Vehicles. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 69–76. [CrossRef]
14. Ferlini, F.; Seman, L.O.; Bezerra, E.A. Enabling ISO 26262 Compliance with Accelerated Diagnostic Coverage Assessment. *Electronics* **2020**, *9*, 732. [CrossRef]
15. da Silva, F.A.; Bagbaba, A.C.; Hamdioui, S.; Sauer, C. Flip Flop Weighting: A technique for estimation of safety metrics in Automotive Designs. In Proceedings of the 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 28–30 June 2021; pp. 1–7. [CrossRef]
16. Raik, J.; Fujiwara, H.; Ubar, R.; Krivenko, A. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In Proceedings of the 2008 17th Asian Test Symposium, Sapporo, Japan, 24–27 November 2008; pp. 21–26. [CrossRef]
17. Syal, M.; Hsiao, M. New techniques for untestable fault identification in sequential circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2006**, *25*, 1117–1131. [CrossRef]
18. Liang, H.C.; Lee, C.L.; Chen, J. Identifying untestable faults in sequential circuits. *IEEE Des. Test Comput.* **1995**, *12*, 14–23. [CrossRef]
19. Condia, J.E.R.; Da Silva, F.A.; Hamdioui, S.; Sauer, C.; Reorda, M.S. Untestable faults identification in GPGPUs for safety-critical applications. In Proceedings of the 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genova, Italy, 27–29 November 2019; pp. 570–573. [CrossRef]
20. Augusto da Silva, F.; Bagbaba, A.C.; Hamdioui, S.; Sauer, C. Combining Fault Analysis Technologies for ISO26262 Functional Safety Verification. In Proceedings of the 2019 IEEE 28th Asian Test Symposium (ATS), Kolkata, India, 10–13 December 2019; pp. 129–1295. [CrossRef]
21. Bernardini, A.; Ecker, W.; Schlichtmann, U. Where formal verification can help in functional safety analysis. In Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 7–10 November 2016; pp. 1–8. [CrossRef]
22. Lai, W.C.; Krstic, A.; Cheng, K.T. Functionally testable path delay faults on a microprocessor. *IEEE Des. Test Comput.* **2000**, *17*, 6–14. [CrossRef]
23. Tille, D.; Drechsler, R. A Fast Untestability Proof for SAT-Based ATPG. In Proceedings of the 2009 DDECS '09—12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, Liberec, Czech Republic, 15–17 April 2009; p. 38–43. [CrossRef]
24. Long, D.E.; Iyer, M.A.; Abramovici, M. FILL and FUNI: Algorithms to Identify Illegal States and Sequentially Untestable Faults. *ACM Trans. Des. Autom. Electron. Syst.* **2000**, *5*, 631–657. [CrossRef]
25. Gursoy, C.; Jenihhin, M.; Oyeniran, A.S.; Piumatti, D.; Raik, J.; Sonza Reorda, M.; Ubar, R. New categories of Safe Faults in a processor-based Embedded System. In Proceedings of the 2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), Cluj-Napoca, Romania, 24–26 April 2019; pp. 1–4. [CrossRef]
26. Cantoro, R.; Carbonara, S.; Floridia, A.; Sanchez, E.; Sonza Reorda, M.; Mess, J.G. Improved Test Solutions for COTS-Based Systems in Space Applications. In *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*; Bombieri, N., Pravadelli, G., Fujita, M., Austin, T., Reis, R., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 187–206.
27. Narang, A.; Venu, B.; Khursheed, S.; Harrod, P. An Exploration of Microprocessor Self-Test Optimisation Based On Safe Faults. In Proceedings of the 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Athens, Greece, 6–8 October 2021; pp. 1–6. [CrossRef]
28. da Silva, F.A.; Bagbaba, A.C.; Sartoni, S.; Cantoro, R.; Sonza Reorda, M.; Hamdioui, S.; Sauer, C. Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs. In Proceedings of the 2020 IEEE European Test Symposium (ETS), Tallinn, Estonia, 25–29 May 2020; pp. 1–6. [CrossRef]
29. Maier, P.R.; Sharif, U.; Mueller-Gritschneder, D.; Schlichtmann, U. Efficient Fault Injection for Embedded Systems: As Fast as Possible but as Accurate as Necessary. In Proceedings of the 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, Spain, 2–4 July 2018; pp. 119–122. [CrossRef]
30. Clarke, E.; McMillan, K.; Campos, S.; Hartonas-Garmhausen, V. Symbolic model checking. In *Computer Aided Verification*; Alur, R., Henzinger, T.A., Eds.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 419–422.
31. Tasiran, S.; Keutzer, K. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test Comput.* **2001**, *18*, 36–45. [CrossRef]
32. Devarajegowda, K.; Servadei, L.; Han, Z.; Werner, M.; Ecker, W. Formal Verification Methodology in an Industrial Setup. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; pp. 610–614. [CrossRef]
33. Lach, J.; Bingham, S.; Elks, C.; Lenhart, T.; Nguyen, T.; Salaun, P. Accessible formal verification for safety-critical hardware design. In Proceedings of the RAMS '06—Annual Reliability and Maintainability Symposium, Newport Beach, CA, USA, 23–26 January 2006; pp. 29–32. [CrossRef]
34. OpenRISC 1000 Architecture Manual. Available online: https://openrisc.io/or1k.html (accessed on 18 December 2021).
35. RTEMS Real Time Operating Systems (RTOS). Available online: https://www.rtems.org/ (accessed on 18 December 2021).

36. Chen, H.; Tian, J. Research on the Controller Area Network. In Proceedings of the 2009 International Conference on Networking and Digital Society, Guiyang, China, 30–31 May 2009; Volume 2, pp. 251–254. [CrossRef]
37. Cantoro, R.; Sartoni, S.; Sonza Reorda, M. In-field Functional Test of CAN Bus Controllers. In Proceedings of the 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 5–8 April 2020; pp. 1–6. [CrossRef]

Curriculum Vitae

1. Personal data

Name	Ahmet Çağrı Bağbaba
Date and place of birth	12.07.1989, Ankara, Turkey
Nationality	Turkish

2. Contact information

Address	Tallinn University of Technology, School of Information Technologies, Department of Computer Systems, Ehitajate tee 5, 19086 Tallinn, Estonia
Phone	+491749156442
E-mail	ahbagb@ttu.ee, acbagbaba@gmail.com

3. Education

2018–2021	Tallinn University of Technology, School of Information Technologies, Computer and System Engineering, PhD studies
2013–2015	Istanbul Technical University, Faculty of Electrical and Electronics Engineering, Electronics Engineering, MSc
2008–2013	Istanbul Technical University, Faculty of Electrical and Electronics Engineering, Electronics and Communication Engineering, BSc

4. Language competence

Turkish	native
English	fluent
German	beginner

5. Professional employment

2021– ...	Cadence Design Systems, Munich, Germany - Application Engineer
2017– 2021	Cadence Design Systems, Munich, Germany - PhD Candidate on Functional Safety
2017	Anka Microelectronic Systems, Istanbul, Turkey - ASIC Design Engineer
2013–2017	Istanbul Technical University, Istanbul, Turkey - Research & Teaching Assistant

6. Computer skills

- Operating systems: Unix, Windows
- Document preparation: Ms-Word, LATEX
- Programming languages: C, C++, Python
- Hardware design languages: Verilog, VHDL, System Verilog

7. Defended theses

- 2015, Leon3-based SoC Implementation on an FPGA, MSc, supervisor Prof. Berna Ors Yalcin, Istanbul Technical University, Institute of Graduate School
- 2013, Secure Near Field Communication System Implementation on an FPGA, supervisor Prof. Berna Ors Yalcin, Istanbul Technical University, Faculty of Electrical and Electronics Engineering

8. Field of research

- Functional Safety
- Digital ASIC Design

Elulookirjeldus

1. Isikuandmed

Nimi	Ahmet Çağrı Bağbaba
Sünniaeg ja -koht	12.07.1989, Ankara, Türgi
Kodakondsus	Türgi

2. Kontaktandmed

Aadress	Tallinna Tehnikaülikool, Arvutisüsteemide Instituut, Ehitajate tee 5, 19086 Tallinn, Estonia
Telefon	+491749156442
E-post	ahbagb@ttu.ee, acbagbaba@gmail.com

3. Haridus

2018–2021	Tallinna Tehnikaülikool, Infotehnoloogia teaduskond, Arvutisüsteemid, Doktoriõpe
2013–2015	Istanbuli Tehnikaülikool, elektri- ja elektroonikatehnika teaduskond, elektroonikatehnika, MSc
2008–2013	Istanbuli Tehnikaülikool, elektri- ja elektroonikatehnika teaduskond, elektroonika- ja sidetehnika, BSc

4. Keelteoskus

türgi keel	emakeel
inglise keel	ladus
saksa keel	algaja

5. Teenistuskäik

2021– ...	Cadence Design Systems, Rakenduse insener
2017– 2021	Cadence Design Systems, PhD kandidaat
2017	Anka Microelectronic Systems, ASIC projekteerimisinsener
2013–2017	Istanbul Technical University, Uurimis- ja õppeassistent

6. Arvutioskus

- Operatsioonisüsteemid: Unix, Windows
- Kontoritarkvara: Ms-Word, LATEX
- Programmeerimiskeeled: C, C++, Python

7. Kaitstud lõputööd

- 2015, Leon3-põhine SoC juurutamine FPGA-l, MSc, juhendaja prof Berna Ors Yalcin, Istanbuli tehnikaülikool, kraadiõppe instituut
- 2013, Turvalise lähiväljasidesüsteemi juurutamine FPGA-l, juhendaja prof Berna Ors Yalcin, Istanbuli tehnikaülikool, Elektri- ja elektroonikatehnika teaduskond

8. Teadustöö põhisuunad

- Funktsionaalse ohutuse kontrollimine
- Digitaalne ASIC disain

ISSN 2585-6901 (PDF)
ISBN 978-9949-83-803-5 (PDF)