Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Advanced Methods for Model-Driven Safety Analysis and Verification

Vom Fachbereich Elektrotechnik und Informationstechnik
der Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
zur Verleihung des akademischen Grades

Doktor der Ingenieurswissenschaften (Dr.-Ing.)

genehmigte Dissertation von

Endri Kaja

D 386

Datum der Einreichung:	26.06.2024
Datum der mündlichen Prüfung:	29.10.2024
Dekan des Fachbereichs:	Prof. Dr. Daniel Görges
Vorsitzender der Prüfungskommission:	Prof. Dr. Gerhard Fohler
Gutachter:	Prof. Dr. Wolfgang Kunz and Prof. Dr. Wolfgang Ecker

Abstract

The semiconductor industry is experiencing rapid growth, which is driving the need for innovative development methodologies, particularly in the digital design domain. In addition to ensuring correct behavior, designers must guarantee that the chips operate at predefined levels of reliability when used in automotive products. Safety considerations have become an integral part of the development process following the adoption of the ISO 26262 safety standard for automotive safety-critical systems. ISO 26262 ensures that these systems behave according to required safety levels by mitigating the risk of hazardous malfunctions. The standard recommends fault injection to verify and analyze safety-critical systems; however, this process often proves to be laborious and error-prone. To combat these challenges, advanced and automated methods are required. Accordingly, this thesis automates the safety verification process, utilizing model-driven architecture principles to enhance productivity, quality, and reliability.

The thesis proposes the generation of mixed-granularity models to represent designs, where gate-level models represent safety-critical components and Register-Transfer Level (RTL) represents the rest of the design. The mixed-granularity model is achieved through model transformation, and formal equivalence checks are applied during this process to ensure the correctness of the transformation. This approach also enables fault injection directly into the design, reducing the overhead caused by the use of additional fault injection tools. The absence of bugs of this process is guaranteed by equivalence check. Furthermore, the ability to inject faults at the design level has enabled the development of a novel fault emulation framework utilizing FPGAs. This proposed framework scales well to large hardware designs and has been successfully applied to several RISC-V based CPU subsystems with a runtime advantage over traditional simulation-based fault injection frameworks. Moreover, an automated flow based on formal methods has been developed that verifies the functionality of design hardening mechanisms and their integration into processor designs. This flow delivers high-quality verification results without requiring white-box design knowledge. Furthermore, the thesis introduces an automated and efficient approach for generating Software-Based Self Tests (SBST) tailored for RISC-V processor cores. This method involves testing processor cores through instructions, providing a viable alternative to hardware-centric solutions. Additionally, the SBST is combined with a novel Program Flow Checking (PFC) technique to achieve high fault coverage and high fault detection rates that comply with ISO 26262 requirements. The proposed PFC is a mixed hardware/software solution that detects faults by monitoring program flow execution.

The proposed solutions have proven to be effective in numerous industrial designs. The achieved results demonstrate that automated approaches significantly reduce development costs and are less susceptible to errors. Through a holistic generation flow, multiple safety verification techniques have been explored, encompassing methods based on simulation, emulation, and formal verification.

Acknowledgements

The work conducted in this thesis was carried out at Infineon Technologies AG, Germany in collaboration with the EDA chair at Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, Germany.

I am profoundly grateful to Prof. Dr. Wolfgang Ecker and Prof. Dr. Wolfgang Kunz for their mentorship and for providing the opportunity to purse this doctoral work. I would also like to thank Alexander Rath, Jürgen Kaspar and Roland Schwenk for giving me the opportunity to conduct my research at their departments.

First and foremost, my deepest appreciation is directed to Prof. Dr. Wolfgang Ecker, whose guidance and support were instrumental throughout my PhD journey. This research work would not be possible without the new ideas that were born during our discussions. I am happy to remain in his research group and further collaborating on novel topics.

My sincerest thanks also go to Prof. Dr. Wolfgang Kunz and Prof. Dr. Dominik Stoffel for their valuable support and insightful recommendations throughout my doctoral research. Their continuous feedback and dedication to reviewing this thesis have been invaluable.

I would like to extend my heartfelt thanks to Dr. Keerthikumara Devarajegowda, under whose guidance my Infineon journey started during my master's studies. I learned a lot from him and his continued support played a significant role in the initial stages of my doctoral research.

I want to thank my PhD colleagues, Nicolas Gerlin and Daniela Sanchez Lopera, for all the time we spent together discussing, planning and finalizing our research.

I wish to thank all my previous students (now colleagues), Mounika Vaddeboina, Luis Rivas, Monideep Bora, Bihan Zhao, and Jad Al Halabi, for their contribution to this thesis. I am indebted to Stephanie Ecker for helping me with the German summary.

I am grateful to all my other colleagues at Infineon for their assistance and encouragement, with a special mention to Ares, Azam, Bryan, Igli, Robert, and Sebastian for their support.

The doctoral research requires not only technical support but also personal support. Therefore, I would like to express my deepest gratitude to Merita for supporting, listening and pushing me to finalize my PhD journey. Also, i would like to acknowledge my good friend Arbër for his continuous motivation.

Lastly, but not least, I hold immense gratitude for my family—my brother Rajner, and my parents Afërdita and Ilir—who have been a constant source of motivation and support throughout every phase of my PhD journey.

Contents

1 Introduction **1**
- 1.1 Safety-Critical Designs . . . 3
- 1.2 Problem Statement and Challenges . . . 4
- 1.3 Requirements . . . 5
- 1.4 Envisioned Approach . . . 7
- 1.5 State-of-the-Art . . . 9
  - 1.5.1 Simulation-Based Fault Injection Techniques . . . 9
  - 1.5.2 Emulation-Based Fault Injection Techniques . . . 12
  - 1.5.3 Formal-Based Fault Injection Techniques . . . 14
  - 1.5.4 Software-Based Self Test Techniques . . . 15
- 1.6 Thesis Overview . . . 16
- 1.7 Publication List . . . 17

2 Model-driven Code Generation Techniques **21**
- 2.1 Metamodeling . . . 21
- 2.2 Metamodel-Based Code Generation . . . 23
- 2.3 Model Transformation . . . 24
- 2.4 Model Driven Architecture . . . 25
  - 2.4.1 MDA Applied to RTL Generation . . . 26
  - 2.4.2 MDA Applied to Properties Generation and 4-eyes Principle . . . 27

3 Functional Safety **31**
- 3.1 Fault Concepts . . . 32
  - 3.1.1 Fault Modeling . . . 32
  - 3.1.2 Fault Testability . . . 33
  - 3.1.3 Fault Collapsing . . . 34
- 3.2 Automotive Safety Standard . . . 35
  - 3.2.1 Automotive Safety Integrity Level . . . 35
  - 3.2.2 Fault Classification . . . 36
  - 3.2.3 Hardware Fault Coverage Metrics . . . 37
- 3.3 Standard-Compliant Safety Design . . . 38
  - 3.3.1 Safety Mechanisms based on Information Redundancy . . . 39
  - 3.3.2 Safety Mechanisms based on Spatial Redundancy . . . 40
- 3.4 Standard-Compliant Safety Verification . . . 41
  - 3.4.1 Overview of the Fault Injection Process . . . 41

3.4.2 Fault Injection Attributes . . . 42
3.4.3 Fault Injection Techniques . . . 43
3.4.4 Formal Verification . . . 47

4 A Generic Approach for Fault Handling 51
4.1 The Generic Fault Handling . . . 52
4.1.1 Specifications layer . . . 53
4.1.2 Model layer . . . 55
4.1.3 View layer . . . 60
4.2 Generic Documentation of Fault Injection Campaigns . . . 62
4.2.1 Overview of the Documentation Generation Framework . . . 62
4.2.2 Fault Injection Documentation Generation . . . 63

5 Fault Simulation on Mixed Granularity RTL Models 65
5.1 Overall Flow . . . 66
5.2 Background on Model Transformation . . . 68
5.3 Generation of Fine-grained Models . . . 69
5.3.1 Netlist-to-ToD . . . 69
5.3.2 Fine-Grained MoD . . . 73
5.4 Fault Injection through Model Transformation . . . 73
5.4.1 Fault Injectors . . . 74
5.4.2 Fault Collapsing . . . 75
5.4.3 Insertion of Fault injectors . . . 75
5.5 Equivalence Checking and Property Checking . . . 77

6 Model-Driven FPGA-Based Fault Emulation 79
6.1 Overview of the Fault Emulator Architecture . . . 80
6.2 Fault Controller . . . 81
6.2.1 Fault Sequencer . . . 82
6.2.2 Fault Decoder . . . 83
6.3 Postprocessing Block . . . 83
6.4 Data Harvesting Logic . . . 85
6.5 Fault Emulation Optimizations . . . 86
6.5.1 Memory Optimization . . . 86
6.5.2 Time Optimization . . . 87

7 Safety Verification of Hardened Processor Cores 89
7.1 Background . . . 90
7.1.1 Safety Transformation Flow . . . 90
7.1.2 Complete Functional Verification of Processor Cores . . . 90
7.1.3 RISC-V CPU Metamodel . . . 92
7.2 Overview of Safety Verification of Processor Cores . . . 93
7.3 Exhaustive Processor Fault Injection . . . 95
7.3.1 Verification Computation Model . . . 95
7.3.2 Fault Model Definition . . . 96
7.4 Formal-Based Fault Propagation Analysis . . . 98

7.4.1 Verification Computation Model . . . 98
7.4.2 Fault Model Definition . . . 100

8 An Automated and Effective Approach for SBST Generation Targeting RISC-V CPUs 101
8.1 Overview of the SBST . . . 102
8.2 Test Pattern Generation . . . 103
8.2.1 DUT and Properties . . . 104
8.2.2 Test Pattern Generation Flow . . . 106
8.3 Program Flow Checking . . . 107
8.3.1 PFC hardware . . . 108
8.3.2 Fault detection flow . . . 108

9 Experimental Results and Discussions 111
9.1 Fine-grained RTL Models Performance . . . 111
9.1.1 Experimental Setup . . . 111
9.1.2 Performance Evaluation . . . 112
9.1.3 Application . . . 114
9.1.4 Discussions and Observations . . . 114
9.2 Analysis and Performance of Fault Emulator . . . 115
9.2.1 Experimental Setup . . . 115
9.2.2 Hardware Utilization . . . 115
9.2.3 Performance Evaluation of the Fault Emulator . . . 118
9.2.4 Performance Evaluation of the "On-the-Fly" Emulation Technique . . . 119
9.2.5 Emulation-based Fault Propagation Analysis . . . 119
9.2.6 Discussions and Observations . . . 120
9.3 Case study: Statistical-based Fault Propagation Analysis . . . 121
9.3.1 Experimental Setup . . . 121
9.3.2 CPU Workloads for Fault Propagation Analysis . . . 121
9.3.3 Fault Propagation Analysis . . . 122
9.4 Analysis of Processor Safety Verification . . . 123
9.4.1 Experimental Setup . . . 123
9.4.2 Processor Hardening Verification . . . 124
9.4.3 Processor Fault Propagation Analysis . . . 125
9.4.4 Discussions and Observations . . . 128
9.5 Analysis of the Automated SBST Generation . . . 129
9.5.1 Experimental Setup . . . 129
9.5.2 SBST results . . . 129

10 Summary of Contributions 131

11 Deutsche Zusammenfassung 135

Bibliography 139

Chapter 1

Introduction

The technological breakthroughs that emerged in the mid-20th century gave birth to one of the most influential sectors in contemporary society, namely, the semiconductor industry. These technological advances have not only revolutionized contemporary technology but have also laid the cornerstone for the digital era. The profound influence of the semiconductor industry has enabled numerous governments to enact legislation aimed at increasing investments in this field. According to McKinsey's projections [10], the worldwide semiconductor industry is expected to reach a market valuation of one trillion dollars by the year 2030. This projected growth is anticipated to impact various sectors, encompassing domains like consumer electronics, automotive, financial services, and many others. In the present era, the majority of electronic components are designed and engineered using a so called System-on-Chip (SoC) design flow. The approach includes the holistic development of comprehensive systems, integrating numerous components. This rapid growth necessitates the development of increasingly complex digital design solutions. As an illustration, today's microprocessors may comprise billions of transistors, underlining the complex nature of these technological advancements.

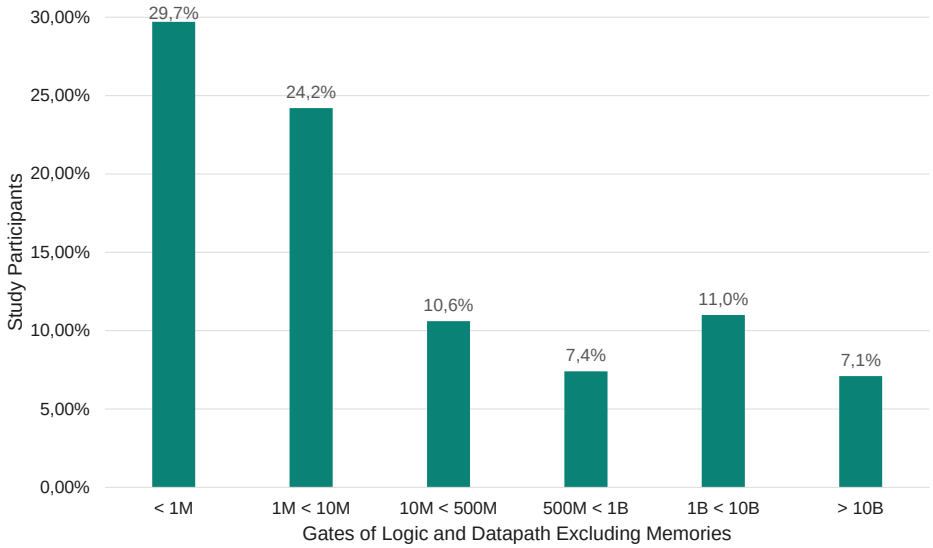


Figure 1.1: IC/ASIC Design Sizes [7]

In 2022, the Wilson Research Group conducted an extensive survey focusing on gate counts

within Application Specific Integrated Circuits (ASICs) [7], as depicted in Figure 1.1. Remarkably, the findings reveal that approximately 36% of ASIC projects exceed 10 million gates, while approximately 7% involve ASICs with 10 billion gates or more. The research [7] study also illustrates the increasing presence of Artificial Intelligence (AI) accelerators and RISC-V based processors within ASICs developed in 2022. To provide a clearer perspective, it reveals that 32% of ASICs now integrate at least one AI accelerator (a notable increase from 27% in 2020). Additionally, approximately 30% of ASICs now feature a RISC-V processor (compared to 23% in 2020).

The rapid and exponential advancement of technologies over the past two decades encompassing AI accelerators, the RISC-V ecosystem, the Internet-of-Things (IoT), Cyber-Physical Systems, Smart Systems, and numerous other innovations has propelled a complex development process. This complex process requires not only smart and automated digital design techniques but at the same time necessitates the application of sophisticated techniques to ensure their correctness. Despite the advancements in semiconductor industry, the process of verification continues to pose a significant challenge, acting as a bottleneck hindering design productivity and compressing time-to-market schedules. This challenge has been widely acknowledged within the industry. Notably, in 2007, the number of design engineers significantly outnumbered verification engineers (about twice as much), whereas, in 2022, approximately 15 years later, these two groups are nearly at parity, with verification engineers holding a slight advantage in numbers, as reported by Wilson Research Group [8] and illustrated in Figure 1.2.

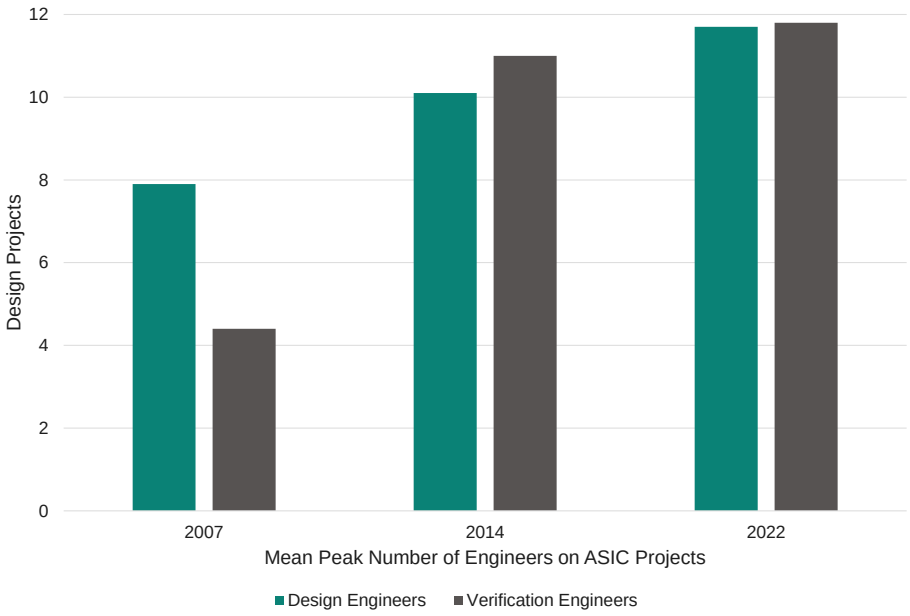


Figure 1.2: Mean Peak Number of Engineers per IC/ASIC Project [8]

This thesis aims to overcome the fundamental verification challenges, with a specific focus on the safety-critical domain. Therefore, it seeks to provide solutions and methodologies to enhance the reliability, accuracy, and performance of verification processes within the safety-critical context.

1.1 Safety-Critical Designs

The rise in design size, as depicted in Figure 1.1, is just a part of the increasing complexity puzzle. Another major factor contributing to the growing complexity in IC/ASIC design and verification is the emergence of additional layers of design prerequisites. These new requirements go beyond design's functionality and include elements like *safety* and *security*. The primary focus of this thesis revolves around safety considerations, which are important in the context of SoC design. Numerous SoCs find application in safety-critical domains, encompassing sectors like automotive, aerospace, healthcare, financial systems, and more. In these environments, the priority is placed on developing systems or products that prioritize safety above all else. These specialized systems are commonly referred to as safety-critical designs, reflecting their critical role in protecting human lives and ensuring the integrity of sensitive operations. Wilson Research Group measures that 44% of ASIC projects have incorporated safety-critical features into their designs [7]. These projects have adhered to different safety standards, as shown in Figure 1.3. Particularly, ISO26262, the automotive safety standard, holds significant importance and is widely applied in various automotive-related SoCs. This standard has an important role in shaping the techniques developed in this thesis.

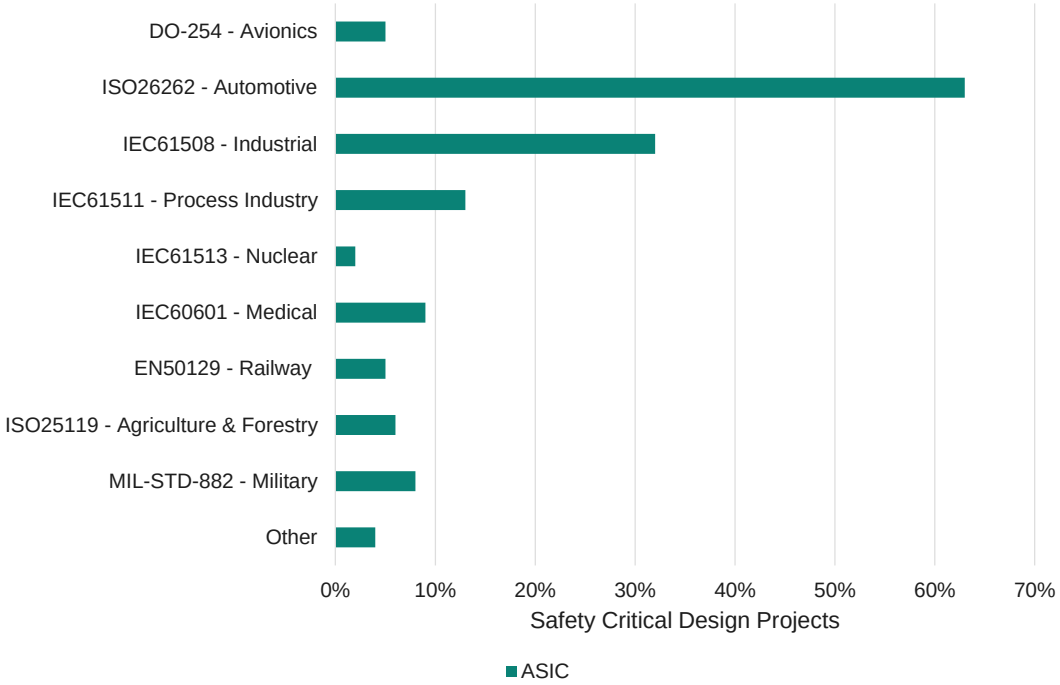


Figure 1.3: Safety-critical development standard used on IC/ASIC project [7]

In safety-critical designs, precautionary measures like safety mechanisms are implemented to protect digital designs against unexpected failures that may arise from faults. These mechanisms serve the purpose of identifying and potentially correcting the effects of such faults. Given the potential catastrophic consequences of a safety-critical design failure, it becomes vital to follow a careful design process, undergo thorough verification, and strictly adhere to the relevant safety standards.

1.2 Problem Statement and Challenges

The increasing complexity of modern designs, coupled with stringent safety standards, creates new challenges in ensuring the reliability of systems. According to [6], in the year 2022, just 24% of the projects were successful in the first attempt, also known as "first silicon success". This implies that the remaining 76% of the projects required 2 or more tapeouts, resulting in a higher cost of wafers and masks. Safety-related design flaws accounted for 5-11% of total respins [6]. Indeed, the development of robust safety verification techniques is essential to prevent any safety-related errors to escape to the very late development phases. While the complexity of design accounts for a significant challenge in both the design and verification processes, it is important to note that the safety verification process is inherently cumbersome and presents multiple challenges, aside from design complexity. ISO26262 guidelines recommends *fault injection* as to verify safety-critical designs, yet this process is challenging and time-intensive. Several of the challenges associated with the verification of safety-critical designs encompass the following factors:

$\mathbb{C}$ 1: **Informal Specifications**

- Safety requirements and specifications are typically written in natural language or in a non-formalized way. This leads to ambiguity, confusion and disagreements between different engineers involved in the development process such as the concept engineer, design engineer and verification engineer.

$\mathbb{C}$ 2: **Human Errors**

- Mistakes or flaws within the system can carry substantial safety consequences. Given that the verification process and data analysis are typically human-driven, the inherent potential for human error exists. This may result in overlooking errors during the verification phase.

$\mathbb{C}$ 3: **Changing Design Requirements**

- Safety-critical designs often undergo evolving design requirements, prompting the need for design modifications. These changes may need adjustments to the existing verification workflows or the introduction of additional verification and analysis steps.

$\mathbb{C}$ 4: **Compliance to Safety Standards**

- It is a considerable challenge to adhere to various safety standards with changing requirements and guidelines. Safety-critical designs encompass multiple components, demanding a rigorous verification process to ensure compliance with the chosen standard. The task of satisfying every requirement outlined in these standards can be a demanding task.

$\mathbb{C}$ 5: **Verification Requirements and Efforts**

- The verification of complex safety-critical designs requires thoroughness, which can be limited by resources and the level of expertise. Consequently, safety verification adopts an iterative strategy, encompassing a range of techniques like analysis, simulation, testing, and validation. Therefore a big challenge is faced in terms of efforts.

$\mathbb{C}$ 6: **Limited accuracy**

- Safety verification can be conducted across various levels of design abstraction. When evaluating a safety-critical system, it is necessary to consider various levels of abstraction, ranging from the overall requirements of the system to the detailed aspects of its implementation, such as transistor or gate-level specifics. However, every level of abstraction usually brings its own set of assumptions, simplifications, and estimations, which could limit the accuracy of the verification process.

$\mathbb{C}$ 7: **Time and Cost Constraints**

- Numerous safety verification methods demand a considerable duration to complete, primarily due to the level of design abstraction at which they are executed, resulting in potential project deadline misses. Furthermore, the requirement for licenses and specialized tools creates a significant obstacle in complying with the predetermined budget.

1.3 Requirements

The automotive sector is expected to have 20% market share of the total semiconductor industry by the end of this decade [10]. Considering this impact, this thesis aims to address the challenges encountered in safety verification, especially in the automotive domain, through the creation of a model-driven framework designed to automatically execute safety verification tasks. Given the complex nature of these challenges, it becomes mandatory to create a precise and well-defined set of requirements capable of mitigating the challenges associated with safety verification. It is imperative to adhere to the requirements listed in the following to create the automated framework:

$\mathbb{R}$ 1: **Formalization**

- Safety verification specifications and requirements shall be clear and unambiguous. These specifications shall be captured via formal models that define the expected behavior of the safety verification technique.

$\mathbb{R}$ 2: **Safety Standards Compliance**

- Safety verification technique shall adhere to the relevant safety standard such as ISO26262. According to the standard, fault injection is mandatory to evaluate safety-critical levels and formal verification shall also be deployed to verify safety-critical components.

$\mathbb{R}$ 3: **Automation**

- The process of safety verification shall be automated to the maximum extent possible to minimize human intervention. Automating the flow enhances the reliability, efficiency, and effectiveness of the safety verification process, thereby reducing the need for manual efforts.

$\mathbb{R}$ 4: **Reusability and Adaptability**

- The safety verification flow shall be designed to be reusable and adaptable to changing design requirements. An automated flow that is well-defined can effectively handle modifications to safety-critical designs. This includes performing an impact analysis and re-verification, as needed, to ensure that the verification process remains thorough and reliable.

$\mathbb{R}$ 5: **Extensibility and Flexibility**

- The safety verification framework shall be designed to be extensible and flexible to accommodate changing requirements and designs. This will enable a flawless integration of the verification process into existing workflows and tools. Furthermore, this will allow for the incorporation of multiple verification techniques, including simulation, emulation, and formal verification, to ensure the reliability and effectiveness of the verification process.

$\mathbb{R}$ 6: **Correctness**

- The automated safety verification framework shall be designed to be correct by construction, meaning that the framework should be developed in a way that prevents the introduction of any bugs or defects that could impact the quality of the verification process.

$\mathbb{R}$ 7: **Data Collection and Analysis**

- The safety verification framework shall provide clear procedures for the collection and analysis of safety-related data during both verification and operation. This will enable the identification and classification of the effects of various faults on the design's behavior, which is essential for ensuring a comprehensive and effective safety verification process.

$\mathbb{R}$ 8: **Structured Documentation**

- Safety verification parameters and results shall be clearly and precisely documented in a structured manner to ensure compliance with safety standards. This structured documentation is necessary to establish traceability and maintain transparency between the safety requirements and the corresponding verification activities.

$\mathbb{R}$ 9: **Resource Scalability**

- The safety verification process shall utilize scalable resources to ensure that the verification process can adapt to dynamic changes, while also ensuring the availability of necessary hardware and software resources. Furthermore, the safety verification process must be designed to comply with the budget constraints of the project.

$\mathbb{R}$ 10: **Accuracy and Reliability**

- The safety verification process shall be accurate, reliable, and effective on various levels of abstraction of the system. The results obtained from the verification process should be precise and dependable, such that risks can be identified and mitigated early in the development process, leading to cost reduction.

$\mathbb{R}$ 11: **Performance**

- Safety verification shall be fast. Fast performance is crucial for ensuring an efficient and effective safety verification process. It is further essential that safety verification is executed during the analysis step. A high-performance verification flow effectively addresses safety challenges while accommodating design scalability and timing constraints, ultimately leading to improved safety results.

1.4 Envisioned Approach

Safety verification is a comprehensive process that requires strict adherence to certain safety requirements to detect and even correct any potential faults that could jeopardize the system. In response to the challenges presented previously, this thesis takes a proactive approach to meet the mentioned requirements, while addressing the presented challenges. The thesis adopts a model-driven methodology, providing automated safety verification solutions across various verification techniques, including simulation, emulation, and formal verification. It is worth noting that all the safety verification methods developed in this thesis align with the fault injection principles outlined in ISO26262. Figure 1.4 illustrates the envisioned approach for safety verification that is addressed in this thesis.

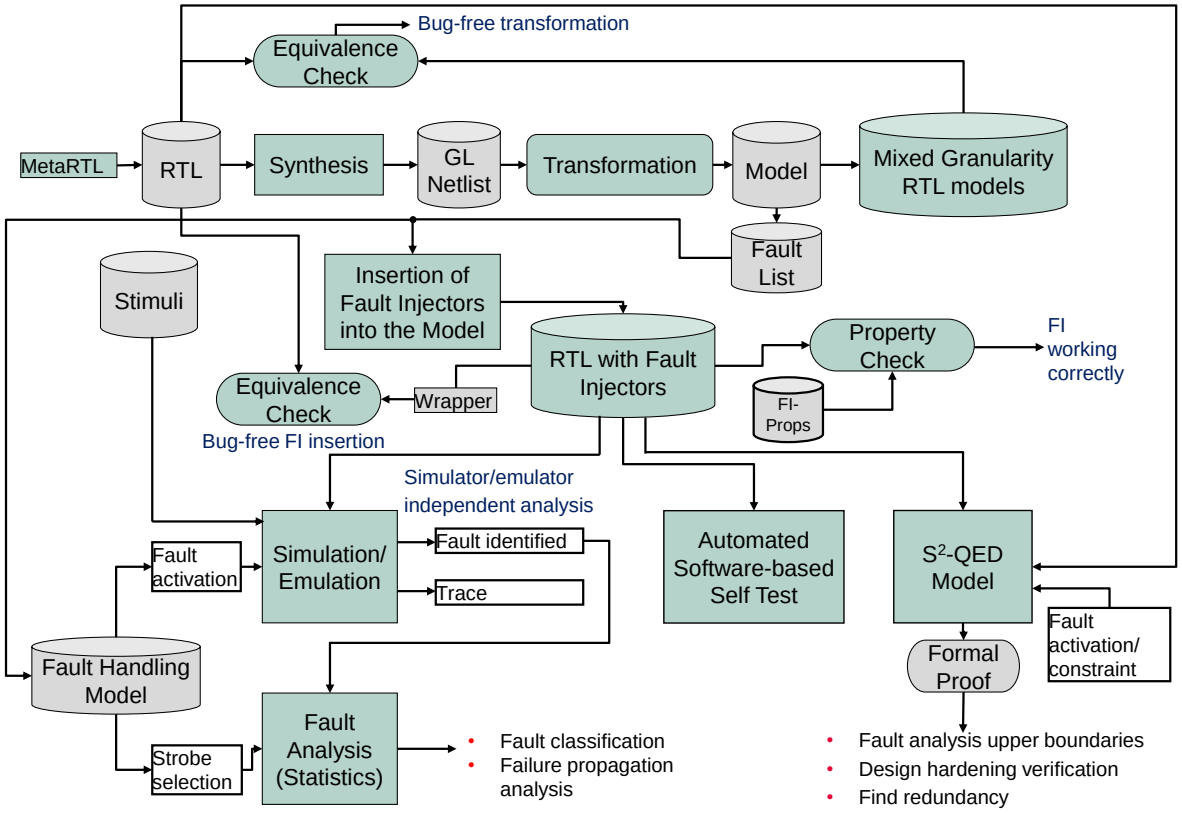


Figure 1.4: Envisioned approach

The proposed methodology incorporates a novel approach, leveraging mixed granularity representation for the Design-under-Test (DUT) to streamline fault injection process. In this

scheme, specific design modules targeted for fault injection are depicted at the gate-level granularity, while the rest of the design is kept in its original Register-Transfer Level (RTL) representation, which remains suitable and accurate for fault propagation analysis. Traditionally, creating RTL design models featuring various gate-level granularity modules is a time-consuming process and prone to errors. To mitigate these challenges, the RTL generation framework, MetaRTL, is integrated into the fault simulation flow. The work introduced in this thesis includes an extension of the existing RTL generation flow by enabling the generation of RTL code at gate-level granularity. MetaRTL is composed of three distinct layers. The specifications are formalized and captured through a metamodel, where the *model* layer, an instance of the metamodel, describes the design independently of the target language. This formalization via the model fulfills requirement $\mathbb{R}1$.

Subsequently, synthesis is performed on the generated RTL to generate the gate-level netlist description of the design. Through a series of transformations, the netlist is reconverted into design models. Each input and output of the gates of the netlist remains visible in the RTL model. Leveraging the MetaRTL framework, RTL models with gate-level granularity are generated, focusing on the component subject to fault injection, while the remainder of the design retains its original RTL granularity. Fault injection on RTL models at gate-level granularity is sufficiently accurate in terms of injected fault models and at the same time speeds up the fault simulation time, thereby meeting requirements $\mathbb{R}10$ and $\mathbb{R}11$. The process of generating RTL models with mixed granularity is fully automated and adaptable to different design variants, aligning with requirements $\mathbb{R}3$ and $\mathbb{R}4$. Following the generation of mixed granularity RTL models, an equivalence check is performed between the original RTL and the mixed version, an essential step to ensure the integrity of the transformation process, thereby ensuring correctness as outlined in requirement $\mathbb{R}6$.

Subsequent stages involve the automated transformation of the mixed granularity design model to incorporate *fault injectors*, enabling the injection of various fault models into the design. The RTL generation framework is employed once more to generate mixed RTL models, including fault injectors, and a second equivalence check is performed, this time with the constraint that fault injectors do not introduce any faults via a wrapper. This is a necessary step to check that model transformation has not introduced any bugs, addressing again requirement $\mathbb{R}6$.

The RTL models, now equipped with fault injectors, are compatible with various open-source and commercial simulation tools, thus fulfilling the resource scalability requirement $\mathbb{R}9$. Moreover, this thesis introduces a novel fault emulation architecture, enhancing fault injection performance while also providing a platform characterized by extensibility and flexibility. This allows the designer to assess a wide range of dependability elements in the initial stages of designs using both simulation and emulation based techniques. As a result, both requirements $\mathbb{R}5$ and $\mathbb{R}11$ are effectively addressed.

The development of the *Fault Handling Model* enables to automatically generate diverse testbenches, each tailored to perform various fault injection campaigns, addressing again requirement $\mathbb{R}3$. Within this model, specific faults are activated according to the chosen fault injection campaign, and the user can select the design strobes (signals) to be analyzed. These testbenches are included within the simulation/emulation framework, and afterwards, a comprehensive analysis is performed to evaluate fault effects. During the fault analysis, structured documentation is generated, offering comprehensive information of faults effects, thereby sat-

isfying requirements $\mathbb{R}7$ and $\mathbb{R}8$.

Additionally, this thesis introduces a novel formal-based approach based in the S^2-QED model, employed to verify the efficacy of design hardening through various safety mechanisms. When performing the technique, it is necessary to activate and constrain the faults according to specifications of safety mechanism. The formal-based technique not only establishes upper boundaries for fault analysis but also identifies potential redundancies within the design, addressing again requirement $\mathbb{R}5$.

Lastly, an automated Software-based Self Test (SBST) has been developed in the scope of this thesis. The SBST takes advantage of formal methods to generate accurate test patterns that provide a high test coverage. Furthermore, the technique is combined with Program Flow Checking (PFC) such that the fault detection rate is aligned with different ASILs from the ISO 26262 standard. The process is fully automated and fulfills various requirements such as $\mathbb{R}1$, $\mathbb{R}2$, $\mathbb{R}3$, $\mathbb{R}4$, $\mathbb{R}5$, $\mathbb{R}10$.

The combination of fault injection processes with diverse safety verification techniques aligns with the recommendations of ISO26262, successfully meeting requirement $\mathbb{R}2$.

1.5 State-of-the-Art

Considerable research efforts have been dedicated to safety verification, with a particular emphasis on fault injection techniques and formal-based approaches for the verification of safety-critical designs. This section provides an overview of state-of-the-art methodologies, starting with simulation and emulation-driven fault-injection techniques and concluding with formal-based methods and SBST.

1.5.1 Simulation-Based Fault Injection Techniques

Fault simulation has traditionally been conducted through two primary methods: the modification of the RTL code or the utilization of the built-in commands provided by the simulator [136]. Below, an overview of the most related works in this field is provided.

Fault simulation via insertion of fault injectors

Fault simulation tools had their origins in the mid-20th century, but it was during the 1990s that they underwent substantial advancements, particularly in fault simulation techniques and tool capabilities. The most famous tools from this era that layed the backbone of many later fault simulation tools can be considered MEFISTO [73] and FERRARI [85].

The fault injection process of MEFISTO involves the introduction of faults into VHDL models through the utilization of specialized fault injectors, which function as probes or saboteurs attached to VHDL signals. The fault injection campaign comprises three distinct phases: a setup phase responsible for generating the executable model and control signals, a simulation phase, and a data processing phase. It is important to note that this tool is exclusively applicable to VHDL models.

FERRARI is a real-time fault injection tool designed with specific objectives. It possesses the capability to inject both temporary and permanent faults, making it suitable for evaluating

the effectiveness of concurrent error detection and correction techniques. Additionally, FERRARI can perform fault injection directly on object code. However, it is essential to note that the tool's coverage depends entirely on the chosen fault models. Furthermore, FERRARI is limited in its ability to inject faults into support logic circuitry, such as memory access control and clock circuitry.

A decade later, significant improvements were done to the VHDL-based fault injection techniques by Baraza et al. [26, 27]. The proposed fault injection techniques encompass the capability to inject both permanent and transient fault models. Fault injection is achieved through automated methods employing mutants and saboteurs within the VFIT tool. This tool reduces the temporal overhead in a large scale compared to the older tools. While this approach offers significant advantages in terms of controllability and observability, it is worth noting that it is confined to injecting faults exclusively at the RTL, which corresponds to VHDL code representations.

Differentiation from related work: *The simulation-based fault injection technique presented in this thesis shares certain similarities with prior works, particularly in the aspect of inserting fault injectors into the design. However, this thesis introduces a novel model-based methodology for inserting fault injectors into the design, all without the need for direct modifications to the RTL representation. Additionally, the entire flow of this thesis is characterized by full automation and is not constrained to VHDL and at the same time provides fault injection on RTL models at gate-level granularity, representing a notable differentiation from previous methodologies.*

Fault simulation via modification of the simulator tool

Many state of the art fault injection methodologies involve direct modifications to the simulator. These modifications include alterations to the simulator's internal architecture, enabling the injection of a diverse range of fault models.

Lee et al. [95] introduce a fault injection method known as SystemC Kernel-based Fault Injection (SyFI). This method involves the modification of the SystemC simulator kernel to enable the injection of various fault models at the SystemC level, which describes the hardware components. SyFI's operation is divided into three distinct phases: setup, execution, and analysis. The authors of this study applied their approach to inject both permanent and transient faults into a processor core based on the MIPS architecture, showcasing its adaptability to different system configurations and fault types.

Ferrareto et al. [61] introduce an automated, non-intrusive fault injection framework that relies on QEMU, an emulator designed for various microprocessor architectures. This framework facilitates the injection of faults into the processor by modifying data structures within QEMU to replicate the fault's behavior. It accommodates various fault types, including stuck-at faults, timing faults, and bit-flips, and has been tested on both x86 and ARM processors. Although this approach offers a rapid simulation workflow for CPU designs, further assessments and evaluations are required to provide a comprehensive understanding of its capabilities.

A methodology for injecting faults into microarchitectural simulators is introduced by [84]. The authors have extended the MARSS and Gem5 simulators to include fault injection capabilities, referred to as MaFIN and GeFIN, respectively. They have implemented a Fault Mask Generator capable of generating random fault masks for different fault types, including bit-

flips, permanent, and intermittent faults. An Injection Campaign Controller is responsible for processing these masks and sending injection requests to the Injector Dispatcher, a module that directly interfaces with the simulators. Experimental evaluations were conducted on both x86 and ARM processors. Similarly, GemFI[108] modifies the Gem5 simulator by introducing additional functionalities to threads to enable the modification of signal run-values.

Differentiation from related work: *In contrast to the approach presented in this thesis, prior research primarily emphasizes the modifications of the simulator's internal structure. However, these approaches are restricted to processor fault injection and have certain accuracy limitations in comparison to cycle-accurate fault injection. The fault injection technique introduced in this thesis is not bound to processor fault injection, and at the same time offers great accuracy by conducting fault injection on RTL models with gate-level granularity.*

Fault simulation on mixed abstraction levels

Concurrent fault simulation on mixed abstraction levels has been researched since the 1980s. Several research works have explored concurrent simulation techniques that combine gate-level and RTL fault simulation. [63] and [97] introduce such techniques, showcasing significant runtime improvements.

Several works, including those by authors in [70, 44, 126], have presented RTL fault modeling techniques that utilize a gate-level representation of the design. The objective of these RTL fault modeling approaches is to create a fault list for the design that accurately represents the stuck-at fault models found at the gate-level. These techniques aim to enhance fault simulation runtime efficiency. However, it's essential to note that they rely on a set of assumptions and estimations to achieve this improvement in efficiency. Espinosa et al. [58] focus on the correlation between RTL and Instruction Set Simulator fault injection. This work demonstrates highly accurate results for permanent fault models by analyzing information derived from executed applications on a microcontroller and approximating fault manifestation probabilities.

Bagbaba et al. [25] introduce a method for the representation of gate-level Single Event Transient (SET) faults through the utilization of multiple Single Event Upset (SEU) faults at the RTL. This technique involves the identification of logic paths associated with each SET within the fan-in logic of flip-flops. The aim is to acquire and subsequently reduce the sets of flip-flops targeted for multiple SEU injections at the RTL level. To assess the impact of these faults, a propagation analysis is conducted employing a formal approach. Experimental outcomes demonstrate a significant reduction in fault spaces, ranging from tens to hundreds of times.

In the scope of fault injection on Virtual Prototypes (VPs), the state of the art and challenges are discussed in [106]. Mueller-Gritschneder et al. [105] use VPs to prepare a processor for fault injection experimentation. The VP serves as a valuable tool to configure the CPU into a fault injection-ready state. Following this preparation phase, the information obtained from the VP is leveraged to initiate a comprehensive RTL simulation of the same processor where the faults will be injected. Tabacaru [125] presents different VP-based fault injection methods by abstracting the gate level information to VP abstraction level. The result showcase 100% correlation between the gate level abstraction and VP. Meanwhile, Cho et al. [41] provide a quantitative evaluation of different soft error injection techniques into VP, RTL and gate-level. The authors have identified disparities in fault-injection outcomes among VPs and RTL

and gate-level models. These discrepancies highlight that error injection techniques at higher abstraction levels may not offer the same degree of precision and accuracy as their counterparts at lower abstraction levels.

Differentiation from related work: *Previous studies have highlighted the benefits of conducting fault injection at different abstraction levels, including gate-level, RTL, and VPs. Nevertheless, many of these studies are constrained to a single simulator, and specific fault models tailored to their respective platforms. Furthermore, most of the studies do not cover all faults. In contrast, the research in this thesis introduces a model-driven fault injection approach that operates independently of the simulator, offering a fully automated and low-effort solution, addressing potential limitations found in previous works.*

1.5.2 Emulation-Based Fault Injection Techniques

Emulation-based fault injection techniques have emerged to accelerate the fault injection process focusing mostly on Field Programmable Gate Arrays (FPGAs)-based fault emulation. Extensive research has been conducted in this domain, and the subsequent section provides an overview of the most related studies to the fault emulator introduced in this thesis.

Fault emulation based on FPGA reconfiguration

FPGA reconfiguration refers to the procedure of altering the FPGA's configuration subsequent to its initial programming [31]. Static FPGA reconfiguration involves the modification of the FPGA's configuration while it is actively processing data. This entails temporarily halting its operation, loading a new configuration, and subsequently resuming its normal functioning. On the other hand, dynamic FPGA reconfiguration, often termed partial reconfiguration, permits the on-the-fly modification of a specific portion of the FPGA's configuration while the remainder of the FPGA design continues to operate without interruption. This approach allows for the real-time adjustment of FPGA sections while ensuring uninterrupted functionality in other areas of the device [9]. This FPGA feature has been utilized to inject faults into the design from several research works.

One of the initial approaches towards fault emulation based on FPGA reconfiguration was presented by [19] but suffered from scalability issues due to incremental synthesis that cased large overheads for large designs. Therefore, recent reseach in this field, as described in [114], leverage Intellectual Property (IP) cores like the AMD Xilinx [132] Soft Error Mitigation Core to inject faults and assess the fault tolerance of designs.

Studies by [96, 65] explore fault emulation techniques leveraging partial FPGA reconfiguration. These research works involve the injection of faults by leveraging partial reconfiguration capabilities offered by FPGAs through the Internal Configuration Access Port (ICAP). The injection of faults is executed by the Microblaze embedded microprocessor, utilizing the HW-ICAP processor core. These frameworks are primarily focused on the introduction of SEU faults specifically into FPGA resources, including Look-Up Tables (LUTs), flip-flops, and Block-RAM components.

Di Carlo et al. [52] employ dynamic partial reconfiguration of FPGA devices, focusing on emulating SEU events within the configuration memory of Xilinx SRAM-based FPGAs. This emulation leverages the Essential Bits technology. A specialized Fault Generator module is

responsible for reading the bitstream configuration file and introducing faults into the design in a pseudo-random manner, considering both the timing and location aspects of the faults. Zhang et al. [135]introduce an alternative method for injecting persistent faults into FPGAs, achieved through the modification of Block RAM (BRAM) configurations.

A fault emulation technique based on run-time reconfiguration (RTR), involving the rewriting of the FPGA's configuration memory, is described in [16, 17]. In this method, faults are physically induced within the FPGA to emulate faults that may occur in the system's model. This emulation process is composed of two distinct phases: first, faults are injected at the specified time, and then they are removed once they cease to exist. The host computer takes charge of the prototyping board via JBits, managing tasks such as reading, analyzing, and generating files for emulating faults. Experiments conducted with three different microcontrollers revealed a primary limitation related to communication bottlenecks, as the host computer oversees the run-time reconfiguration process.

Differentiation from related work: *Prior studies have underscored the enhanced performance achieved in the entire fault injection process through the utilization of FPGA-based emulators. Contrary to this thesis, the previous studies primarily rely on vendor-specific IPs and technologies, making them non-generic for use on diverse fault emulation platforms. Additionally, many of these prior works would benefit from extensions to accommodate a broader range of fault models, with particular emphasis on addressing stuck-at faults.*

Fault emulation based on the insertion of fault injectors

Fault emulation through the insertion of fault injectors presents an alternative to FPGA reconfiguration techniques and increase the controllability of the fault emulation campaigns.

Lopez-Ongil et al. [102] have introduced an autonomous fault emulation framework based on FPGAs, designed to inject SEU faults with the assistance of a host computer. This fault emulation framework encompasses several key components, including an emulation controller, a fault injection module, a fault classification module, a testbench application module, an onboard RAM, and an interface module for communication with the host computer. The fault injection process involves the replacement of flip-flops with mutants and saboteurs, employing various replacement techniques. Notably, experimental outcomes indicate that this system can achieve the execution of over one million faults per second. Entrena et al. [57] extends the framework by proposing a multilevel fault emulation technique. The design, which is target for fault injection, is represented using two independent and identical models at both the gate-level and RTL. This framework allows for switching between these two models during fault emulation. The key advantage of this approach is that the gate-level model offers higher accuracy in fault injection campaigns, enabling a precise assessment of error sensitivity.

Grinschgl et al. [68] introduce an emulation-based fault injection method with a particular emphasis on automating fault injector placement. This technique involves parsing the VHDL description of the design to extract essential signal-related data, which is then stored in Extensible Markup Language (XML) files. Through a Graphical User Interface (GUI), the user selects the specific signal or port where the saboteur needs to be inserted. Subsequently, the fault injectors are routed to the fault injection controller to apply the appropriate stimuli. Sau et al. [115] present a versatile FPGA-based fault injection tool, known as SCHIFI, designed to introduce soft errors within the memory hierarchy of a specified system. The approach involves

the inclusion of saboteurs to selectively flip bits in the memory data where desired, and a Finite State Machine (FSM) is employed to control and manage the injection process meticulously.

Differentiation from related work: *The prior studies have showcased a great degree of controllability in fault emulation campaigns, yet they are encumbered by limitations such as being confined to a single RTL language, like VHDL, restricting fault injection solely to memory cells, or constraining themselves to specific fault models. In contrast, the fault emulator developed in this thesis operates without any of these aforementioned constraints.*

1.5.3 Formal-Based Fault Injection Techniques

The significant drawback of fault simulation and emulation lies in its restricted fault coverage, as it can only identify faults that have been explicitly modeled. Furthermore, the sensitization of faults is dependent on the input stimuli, which means that certain faults may remain undetected, posing the risk of potential design failures. Therefore, exhaustive fault injection techniques have been developed to verify safety-critical designs.

Seshia et al. [120] have introduced a soft error resilience technique that utilizes exhaustive fault injection. This fault injection technique is supplementary to an existing functional verification process. It involves precomputing a comprehensive list of all design latches and translates the SEU formalism into n FSMs, where n corresponds to the number of latches. Initially, a single run of a formal verification tool is executed to assess the design's functional correctness under fault-free conditions. Subsequently, n tool runs are conducted, with one injection targeting each latch per run. If a run fails, it signifies that the corresponding latch requires protection against faults; otherwise, it does not. In a similar context, Krautz et al. [91] have described a formal fault injection technique that evaluates the efficacy of fault-tolerant designs by conducting exhaustive fault injection on fortified designs and quantifying their fault coverage.

Sauer et al. [116, 117] have introduced a SAT-based Automatic Test Pattern Generation (ATPG) methodology, which provides exhaustive tests for detecting delay faults through sensitizing all paths within the design. Diverging from alternative methods, such as the identification of k-longest testable paths [112], the authors consider both upper and lower path length limits, resulting in an extensive set of tests encompassing all sensitizable paths. In line with [33] and [107], the objective of these approaches is to enhance fault coverage by employing multiple test patterns. Conversely, Lingappan et al. [100] generate test patterns at the RTL level itself, achieving substantial fault coverage.

In [23] and [46], the authors employ a combination of fault simulation, ATPG, and formal techniques to enhance the reliability of fault analysis. Initially, the ATPG tool is employed to generate the test pattern. Subsequently, fault simulation is executed to validate the design's functionality under fault conditions. Formal techniques come into play for the identification of untestable and uncovered faults. Finally, the results stemming from formal methods and fault simulation (augmented by ATPG) are subjected to a comparative analysis.

In the study by Fujita et al. [15], the authors introduce an ATPG methodology specifically designed for sequential designs. Their approach initiates by extracting a FSM coupled with a datapath model from the RTL code to acquire output and polynomial functions. This methodology includes two distinct fault types, namely bit failure and condition failure, which are subsequently modeled. The process proceeds with a propagation and justification phase similar to time frame expansion. Ultimately, the high-level test pattern is converted into a gate-level

pattern, and fault simulation is performed on a variety of benchmark circuits.

Differentiation from related work: *In contrast to the approach introduced in this thesis, the previously mentioned techniques do not adhere to a model-driven workflow. Instead, they rely on design-specific white-box information, which may require a high manual effort when transitioning to a different design architecture. Furthermore, for approach presented in this thesis, there is no requirement for test pattern generation; instead, a formal tool issues specific instructions to identify faults. Notably, the technique proposed in this thesis can be applied both; before and after the design undergoes hardening processes.*

1.5.4 Software-Based Self Test Techniques

Design-for-Test (DFT) infrastructures often result in significant overhead in terms of area and performance, which has led to the emergence of various low-cost alternatives, such as SBST techniques. These SBST techniques focus on CPU-based designs and are advantageous in their ability to provide acceptable fault coverage with minimal penalties, while also providing at-speed testing capabilities. In the following sections, the related approaches are described that are most relevant to the SBST technique developed in this thesis.

Chen et al. [39, 40] propose an SBST methodology that utilizes a software tester directly embedded in the CPU memory to perform structural testing. The tester generates pseudo-random patterns that can be easily modified to achieve a higher fault coverage due to software flexibility. The technique initially generates structural test patterns for the components of the CPU rather than considering the entire CPU, and then utilizes instructions at the processor level to test these specific components. Consequently, the random pattern generation is combined with a structural analysis of the components to achieve a high fault coverage. In contrast to random patterns, Paschalis et al. [110] presented a deterministic methodology for SBST. The authors considered the functional modules of the processor datapath and generated test routines by utilizing existing arithmetic operations. In the case of multiplier-accumulator testing, the test routines were generated based on repetitive patterns. The authors demonstrated that the same test routine of repetitive patterns can achieve high fault coverage for any standard array multiplier.

Kranitis et al. [90] propose a high-level SBST methodology to test embedded processors with the main objective of achieving a high structural fault coverage with low test development cost. The authors adopted a divide-and-conquer approach by developing component-based tests. The test development process is based on the processor's instruction set architecture (ISA) and divided into three main phases. Firstly, the identification of processor components and operations takes place. Secondly, similar processor components and operations are classified. Finally, test routines are developed by reusing a test library that focuses on the processor's ISA. The methodology was applied to two different RISC-based processors, achieving high fault coverage of 95% and 92%, respectively. While most SBST methodologies focus on the programmer-visible components of processors, such as the Arithmetic-Logic Unit (ALU), Gizopoulos et al. [66] shifted their attention to testing the pipelining logic of the processors, such as forwarding logic. This methodology extends existing SBST approaches by incorporating extra test patterns to target hazard detection mechanisms, forwarding mechanisms, and address-related components.

Riefert et al. [113] and Faller et al. [60] employ formal methods to generate test programs

for mid-sized processors. To check the design's structural testability, a SAT-solver was utilized. Additionally, the Bounded Model Checking (BMC) approach, using Craig Interpolation prover (CIP) [94], was employed to check the design's functional testability and generate test sequences. Furthermore, Riefert et al. [113] introduced a Validity Checker Module (VCM) that provides a set of functional constraints for the SBST generation. The VCM is a circuit specified using a Hardware Description Language (HDL) and it is combined with the DUT, embedding functional test constraints on the DUT itself. Faller et al. [60] extended the VCM to enable its reusability for various processor families. These methodologies were proven to achieve high fault coverage and improve the testability of the designs.

A more comprehensive and a detailed description of various SBST state-of-the-art methodologies can be found at [111].

Differentiation from related work: *The SBST methodology developed in this thesis distinguishes itself from related works by introducing a fully automated approach that follows a structural model-driven strategy, thereby eliminating the need for manual effort and extensive knowledge of the processor, as required in [90, 66, 39, 40]. Like the SBST methodologies presented in [113, 60, 90, 110], the proposed approach is deterministic, which ensures that the test effectiveness does not depend on the quality of pseudo-random generated tests. The methodologies presented by Riefert at al. [113] and Faller et al. [60] are the most similar to the SBST methodology that is developed in this thesis. However, there are some distinct differences between the methodologies. Specifically, in [113, 60], the SBST constraints are expressed using HDL and embedded in the design, while the proposed methodology uses property-based constraints when generating test patterns, resulting in a reduced hardware footprint. Additionally, most of the existing methodologies only allow memory content observation after the test program execution, thus enabling fault detection only in the later stage. Conversely, the PFC module used in this thesis enables on-the-fly fault detection, meaning that the fault is immediately detected once it propagates to the selected outputs, without the need for external memory units.*

1.6 Thesis Overview

This thesis contributes to the goal of advancing safety verification techniques within the context of complex safety-critical digital designs. The thesis includes in the following nine chapters and the organizational structure of these chapters is outlined as follows:

Chapter 2 offers a comprehensive overview of existing model-driven code generation techniques. This chapter introduces key concepts related to metamodeling and model transformation. The chapter concludes with the definition of the Model-driven Architecture (MDA) and its practical applications in RTL and properties generation.

Chapter 3 serves as an important foundation for this thesis, providing in-depth insights into functional safety. It describes fundamental fault concepts, delves into common safety standards, and illustrates the principles of safety design and verification in alignment with these standards.

Chapter 4 presents a generic approach to fault handling, encompassing two primary parts. The first part describes the model-based fault handling framework, offering a range of algorithms for fault handling. The second part provides an exhaustive exploration of the documentation generation framework.

Chapter 5 discusses the fault simulation process applied to mixed granularity models. It offers a comprehensive description of the entire flow, covering aspects such as model transformation. The chapter also includes illustrative snapshots of the generated code and concludes with the verification of the flow itself, accomplished through equivalence checking and property checking.

Chapter 6 introduces the novel architecture of the model-driven fault emulation framework in detail. It provides a comprehensive explanation of the hardware components constituting the architecture, all of which facilitate the fault emulation process.

Chapter 7 presents a detailed description of the safety verification technique applied to hardened processor cores. This technique draws from the S^2-QED verification model and is automatically generated via the MetaProp property generation framework. Additionally, this chapter provides a description of a formal-based fault propagation analysis.

Chapter 8 provides a detailed description of the SBST generation technique, which has been combined with the PFC approach. The chapter starts with the explanation of the test pattern generation procedure employing formal methods, followed by an a short description on how fault simulation is utilized for the purposes of validation and fault dropping. Afterwards, the chapter describes the PFC implementation and displays how it is employed to detect faults.

Chapter 9 presents results and offers a demonstration of the applicability and effectiveness of the proposed safety verification techniques. It encompasses various parts, including the effectiveness of fault simulation on mixed granularity models, the performance enhancements realized through the fault emulator, statistical fault propagation analysis applied to different RISC-V variants, the verification of hardened processors, and the effectiveness of the automated SBST methodology.

Chapter 10 concludes this thesis, providing a summary of the main contributions and insights gained from this research.

1.7 Publication List

A large part of this thesis, including the developed techniques as well as the snippets of related work, has already been published in the publications listed chronologically as follows:

1. V. B. BAVACHE, Z. HAN, H. HARTLIEB, E. KAJA, K. DEVARAJEGOWDA AND W. ECKER, Automated SoC Hardening with Model Transformation. In *17th Biennial Baltic Electronics Conference (BEC)*, Tallinn, Estonia, 2020, pp. 1-6, (see also [30]).
2. E. KAJA, N. O. LEON, M. WERNER, B. ANDREI-TABACARU, K. DEVARAJEGOWDA AND W. ECKER, Extending Verilator to Enable Fault Simulation. In *MBMV 2021; 24th Workshop*, online, 2021, pp. 1-6, (see also [83]).
3. E. KAJA, N. GERLIN, M. VADDEBOINA, L. RIVAS, S. PREBECK, Z. HAN, K. DEVARAJEGOWDA, AND W. ECKER Towards Fault Simulation at Mixed Register-Transfer / Gate-Level Models. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Athens, Greece, 2021, pp. 1-6, (see also [79]) .
4. K. DEVARAJEGOWDA, E. KAJA, S. PREBECK AND W. ECKER, ISA Modeling with Trace Notation for Context Free Property Generation. In *58th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, 2021, pp. 619-624, (see also

[47]).

5. E. KAJA, N. GERLIN, M. BORA, K. DEVARAJEGOWDA, D. STOFFEL, W. KUNZ AND W. ECKER, MetaFS: Model-driven Fault Simulation Framework. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Austin, TX, USA, 2022, pp. 1-4, (see also [74]).
6. E. KAJA, N. GERLIN, M. BORA, G. RUTSCH, K. DEVARAJEGOWDA, D. STOFFEL, W. KUNZ AND W. ECKER, Fast and Accurate Model-Driven FPGA-based System-Level Fault Emulation. In *IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, Patras, Greece, 2022, pp. 1-6, (see also [76])
7. N. GERLIN, E. KAJA, M. BORA, K. DEVARAJEGOWDA, D. STOFFEL, W. KUNZ AND W. ECKER, Design of a Tightly-Coupled RISC-V Physical Memory Protection Unit for Online Error Detection. In *IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)*, Patras, Greece, 2022, pp. 1-6, (see also [64]).
8. E. KAJA, N. GERLIN, D. STOFFEL, W. KUNZ AND W. ECKER, Automated Thread Evaluation of Various RISC-V Alternatives using Random Instruction Generators. In *proceedings of the Design and Verification Conference and Exhibition (DVCon)*, San Jose, California, United States, 2023, (see also [78]).
9. N. GERLIN, E. KAJA, F. VARGAS, L. LU, A. BREITENREITER, J. CHEN, M. ULBRICHT, M. GOMEZ, A. TAHIRAGA, S. PREBECK, E. JENTZSCH, M. KRSTIĆ AND W. ECKER, Bits, Flips and RISCs. In *26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Tallinn, Estonia, 2023, pp. 140-149, (see also [64]).
10. M. VADDEBOINA, E. KAJA, A. YILMAZER, S. PREBECK AND W. ECKER, Parallel Golomb-Rice Decoder with 8-bit Unary Decoding for Weight Compression in TinyML Applications. In *26th Euromicro Conference Series on Digital System Design (DSD)*, Durres, Albania, 2023, pp. 1-6, (see also [129]).
11. E. KAJA, N. GERLIN, R. KUNZELMANN, K. DEVARAJEGOWDA AND W. ECKER, Modelling Peripheral Designs using FSM-like Notation for Complete Property Set Generation. In *IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, Singapore, 2023, pp. 508-515, (see also [77]).
12. E. KAJA, N. GERLIN, U. YUN, J. AL HALABI, S. PREBECK, D. STOFFEL, W. KUNZ AND W. ECKER, A Statistical and Model-Driven Approach for Comprehensive Fault Propagation Analysis of RISC-V Variants. In *proceedings of the Design and Verification Conference and Exhibition (DVCon)*, San Jose, California, United States, 2024.
13. E. KAJA, N. GERLIN, B. ZHAO, D. SANCHEZ LOPERA, J. AL HALABI, A. SHER KHAN, S. PREBECK, D. STOFFEL, W. KUNZ AND W. ECKER, An Automated Exhaustive Fault Analysis Technique guided by Processor Formal Verification Methods. In *25th International Symposium on Quality Electronic Design*, San Francisco, California, United States, 2024, pp. 1-8, (see also [81]).
14. D. SANCHEZ LOPERA, R. KUNZELMANN, E. KAJA AND W. ECKER, Fake Timer: An Engine for Accurate Timing Estimation in Register Transfer Level Designs. In *25th International Symposium on Quality Electronic Design*, San Francisco, California, United States, 2024, pp. 1-8, (see also [101]).
15. M. VADDEBOINA, E. KAJA, A. YILMAZER, U. GHOSH AND W. ECKER, PaGoRi:A Scalable Parallel Golomb-Rice Decoder. In *27th International Symposium on Design &*

Diagnostics of Electronic Circuits & Systems (DDECS), Kielce, Poland, 2024, pp. 67-72, (see also [130]).

16. E. KAJA, N. GERLIN, J. AL HALABI, A. TAHIRAGA, S. PREBECK, D. STOFFEL, W. KUNZ AND W. ECKER, An Automated and Effective Approach for SBST Generation Targeting RISC-V CPUs. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oxford, United Kingdom, 2024, [ACCEPTED] .
17. E. KAJA, N. GERLIN, A. TAHIRAGA, J. AL HALABI, S. PREBECK, D. STOFFEL, W. KUNZ AND W. ECKER, Special Session: A mixed simulation-, emulation-, and formal-based fault analysis methodology for RISC-V. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oxford, United Kingdom, 2024, [ACCEPTED] .

Chapter 2

Model-driven Code Generation Techniques

Throughout the years, models have been utilized in various technical disciplines, such as engineering, mathematics and science. A model is a formal mathematical construct that represents the system's characteristics, structure and behaviour. The famous statistician George E.P. Box stated that "all models are wrong, but some are useful" [35]. It is practically infeasible to accurately model systems due to the complexity, uncertainty and randomness of the reality. Nevertheless, models have proven themselves to be useful approximations that simplify the formalization, characterization, abstraction and visualization of the system.

In software engineering domain, models are considered as centric artifacts during the software development process. This concept is widely known as Model Driven Engineering (MDE). MDE promotes and facilitates the reuse of the models, thus enabling the automation of repetitive tasks. By using models as blueprints, engineers can streamline their development flow and generate code from models, thus reducing the semantic gap between abstract requirements and final target code. As a result, productivity is enhanced while shortening development time. Furthermore, MDE enables early system verification and validation, allowing developers to find potential errors/problems before the actual implementation by simulating, analyzing and testing models.

MDE-based approaches encompass a range of techniques including metamodeling, model transformation and code generation. The aforementioned techniques are prevalent in safety verification and analysis methods developed in this thesis, therefore this chapter gives a general description of the fundamental concepts for reason of self-containment. In the following, a basic explanation of metamodeling concepts is given. The chapter concludes with MDA and its applications.

2.1 Metamodeling

Metamodeling is an essential element in MDE that provides means to define the structure, semantics and the relationships of the models. Metamodels establish the guidelines and constraints that model artifacts must follow, ensuring their adherence to predefined rules and enabling their organization and classification [54]. By employing metamodeling, MDE enhances the generation of executable code from models utilizing formalized specifications, hence, im-

proving the efficiency and effectiveness of the development process. The term *"meta"* means "after" or "beyond" implying that metamodeling literally represents *modeling models*. In contrast to a model, a metamodel serves as an abstract depiction or representation of the model itself [67]. As a result, there is a hierarchical relation between the system (e.g., hardware circuit), model and metamodel, i.e., the system is an instance of the model, and the model is an instance of the metamodel. Figure 2.1 displays this hierarchical relation by selecting Unified Modeling Language (UML) class diagrams to represent the metamodel, as they offer the capability to encompass attributes and establish relationships at both the class and instance levels through logical connections such as *aggregation, composition, association*.

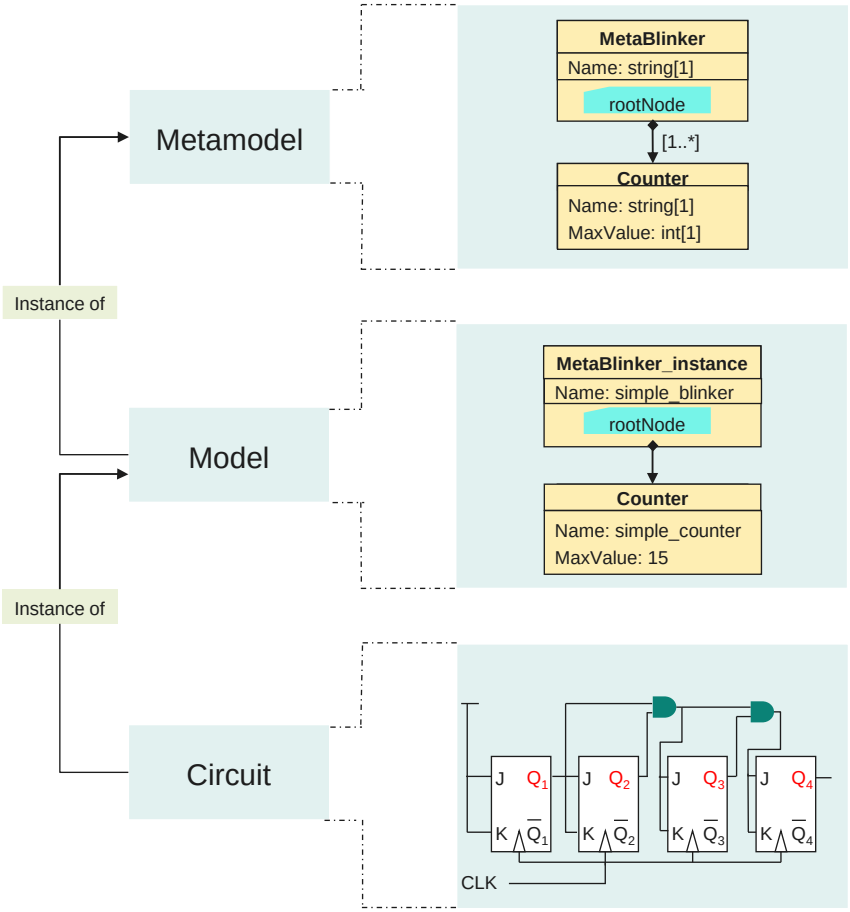


Figure 2.1: Hierarchy between a circuit, a model and a metamodel

The metamodel (right side of Figure 2.1) models a simple hardware blinker by utilizing only two UML classes. The root node is called *MetaBlinker* and has only *Name* as an attribute of type *string*. Generally, all attributes have a type (e.g., string, int, float) and a multiplicity value that defines the number of attribute instances. MetaBlinker has a composition relation to the class *Counter* with a multiplicity of 1..*, meaning that one to many *Counter* instances can be associated with MetaBlinker. The *Counter* class has its own attributes such as *Name* and *MaxValue* that simply determine the properties of a counter circuit.

As a next step, the model instance is created that complies with metamodel rules and definitions. An illustrative example of the instance is depicted in the figure, showcasing a straight-

forward blinker system featuring a single counter. The maximum count value for this counter is set at 15. Finally, the model instance, which represents the actual circuit, is presented. The figure illustrates the design of the counter, consisting of four interconnected JK flip-flops. This configuration enables the counter to count up to a maximum value of 15.

Clearly, each instance complies with the constraints set by the higher layer of the hierarchy. Specifically, the model instance adheres to the requirements outlined by the metamodel, while the circuit itself adheres to the specifications established by the model.

2.2 Metamodel-Based Code Generation

Metamodeling finds a wide usage both in the hardware and software engineering domains because it enables code generation frameworks to automatically produce the intended code artifacts in a decoupled manner from specific languages or platforms. Also, metamodeling offers a high degree of flexibility since only metamodel modifications are required to enable new features of code generation. The numerous listed benefits offer a great potential to increase the productivity and reduce the overall Time-to-Market.

Following the metamodeling principles, an Infineon in-house framework, known as *Metagen*, is widely utilized to increase the design productivity, enhance design quality, and reduce turnaround time [55]. Additionally, metamodel-based code generators have increased the productivity of single design steps by a factor of 20x and implementations of chips by a factor of 3x [55].

The generation flow of the Metagen framework is displayed in Figure 2.2. The centric element of the framework is the metamodel that captures system's requirements and specifications.

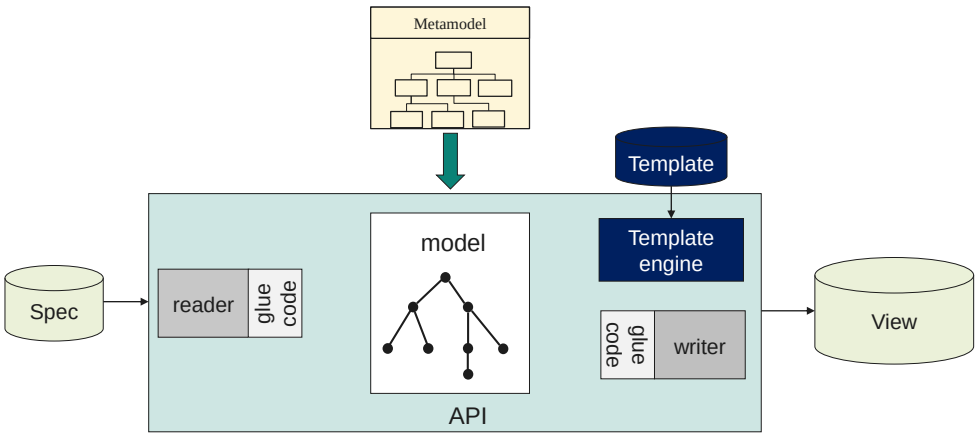


Figure 2.2: Metagen Framework

As described earlier, the metamodeling concepts facilitate the framework's capability to read and write models conforming to their predefined metamodel. As can be seen in the figure, the metamodel is defined via UML class diagrams. Accordingly, a Python-based infrastructure is generated based on the metamodel definitions, i.e., the generated Application Program Interfaces (APIs) provide various methods adhering to metamodel objects, their properties and the relation between them. These API methods allow to create, read and modify the model.

Initially, the user utilizes a GUI tool provided by the framework to create model instances (specifications). Subsequently, an auto-generated *reader* leverages the framework's API to generate the model by extracting and interpreting the specifications. These specifications can be presented in various document formats, including XLS, XML, or other formalisms. The *glue code* is manually written in Python that enhances the reader and enables further customizations to the model. Clearly, the created model provides data needed by the generators in a structured way. In the subsequent stage, the code generator is employed to transform the model into code. The code generator may be a template engine (as depicted in the figure) or an standalone program writing to a file targeting a specific code structure. The template engine allows to efficiently mix code pieces, generate pragmas and to produce the desired final code according to the data provided in the model by using *Mako* templates. Additionally, *writer* templates (built-in or manually written) utilize the API to access the model and facilitate the generation of code according to a predefined format. Lastly, the *View* is generated. A *view* is an ASCII text which represents e.g., XML documents, C Code (Programs), VHDL, Verilog or similar formalisms.

Relevance to this thesis

The metamodel-based code generation framework provides numerous advantages, with a key benefit being its ability to generate code in a structured manner. This thesis extensively utilizes Metagen to establish a versatile and adaptable approach for managing various fault injection campaigns, requiring minimal manual efforts.

2.3 Model Transformation

Model transformations are a key element in MDE that allows for the manipulation and refinement of repetitive tasks in an automated manner. The Object Management Group (OMG) [5] defines model transformation as the process of converting one or multiple models, i.e., source models, to one output model, i.e., target model, of the same system. There are three types of model transformation widely used in MDE: Model-to-Model (M2M), Model-to-Artifact (M2A), and Artifact-to-Model (A2M), where artifact refers to general textual artifacts such as code or documentation.

Generally, M2M transformation is defined at different levels such as metamodel, model, and instance. This definition is given by Koch [87] as following: At the metamodel level, a transformation involves specifying certain types of source models that are converted into a different type of target models; at the model level, a transformation entails identifying specific model elements that will be transformed based on predefined rules; at the instance level, a transformation involves identifying specific objects within the model that will be transformed in a particular manner.

A high level view of model transformation flow is presented in Figure 2.3. Transformation rules or mappings are established to define how elements, relationships, and properties from the source model are converted into the target model. The transformation engine adheres to these rules and maps the source model to the target model. Usually, these transformations primarily follow a vertical orientation by refining a view or model from a higher level of abstraction to a lower level, incorporating more detailed information and implementation specifics.

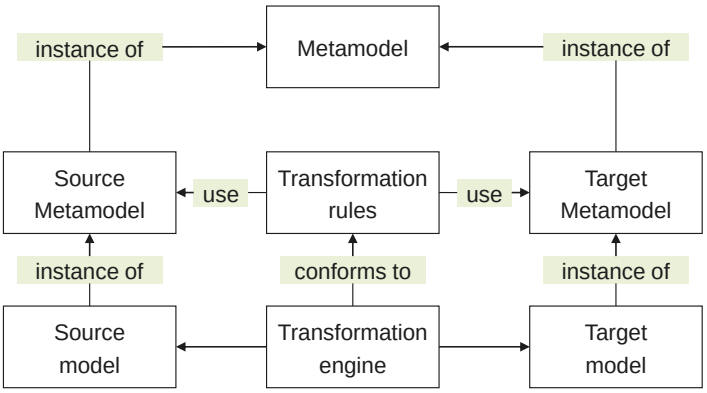


Figure 2.3: Model transformation flow [37]

Relevance to this thesis

This thesis extensively employs a Model-to-Model (M2M) transformation to equip the original design model with fault injection capabilities. The utilization of M2M transformation greatly automates the process and promotes reusability, significantly reducing manual efforts and minimizing the risk of human errors.

2.4 Model Driven Architecture

MDA is a specialized subset of the broader MDE concept. MDA focuses on providing a structured framework for system development that revolves around the use of models and emphasizes platform independence. In contrast, MDE encompasses a wider array of modeling techniques and tools that can be applied throughout different stages of the system development lifecycle.

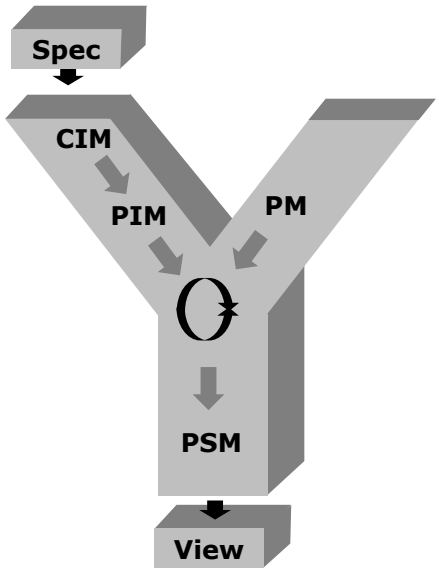


Figure 2.4: MDA principles in Y-Chart [4]

MDA, a concept defined by OMG [5], partitions the system into three consecutive layers.

The Y-chart in Figure 2.4 illustrates the stack of these layers. The MDA approach consists of the following layers [45]:

- **Computation Independent Model (CIM)** represents the design specifications. This top-level model, while comprehensive, does not contain specific implementation details or architectural information.
- **Platform Independent Model (PIM)** follows the top-level design specifications and defines both the implementation and architecture of the system. However, it conceals the details related to the specific usage of the concrete platform.
- **Platform Specific Model (PSM)**, commonly referred to as a view model, closely resembles the generated code. It is the model that exhibits the highest level of correspondence to the final code artifacts.
- **Platform Model (PM)** gathers information from the PIM and is responsible for defining the specific platform that will be utilized for the system implementation. This model connects the abstract representation in the PIM and the concrete platform to be employed. PM is considered at the same level as PSM.

These layers help transforming high-level diagrams and models into executable code and act as a bridge between the system concept and the final implementation. Due to the clear advantages, the combination of MDA concepts with Metagen helps to overcome challenges of developing code generators and closing the semantic gap between the specifications/requirements and the final generated system. The following subsections provide an overview of RTL and Properties Generation, both adhering to the principles of MDA.

2.4.1 MDA Applied to RTL Generation

MDA principles are exploited by Ecker et al. [118, 53] to create a model-driven framework, namely MetaRTL, that generates RTL from specifications in a structured way. The main idea of MetaRTL is to encourage the design-centric development of RTL and to let the generator backend handle the simulation aspects of the design. The RTL generation flow is illustrated in Figure 2.5. Similarly to MDA, the flow consists of three layers such as *Model-of-Things (MoT)*, *Model-of-Design (MoD)*, and *Model-of-View (MoV)* and each layer contains its own model.

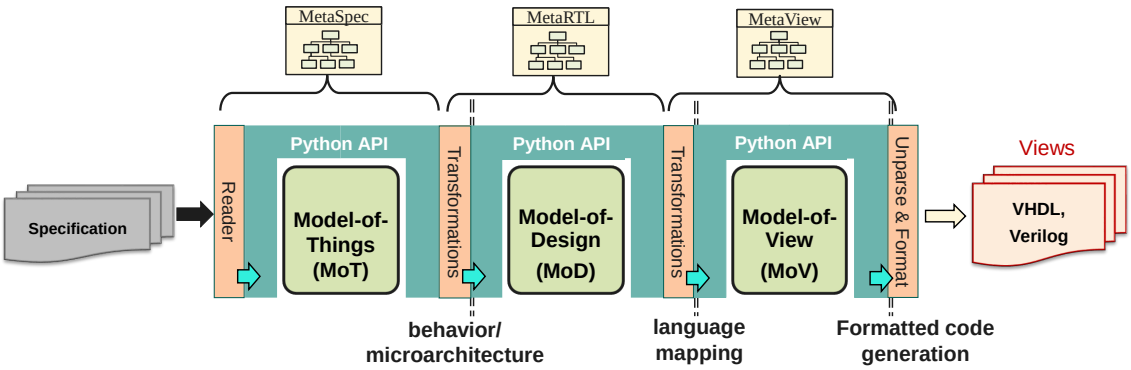


Figure 2.5: RTL generation flow

At first, the process involves a *reader* translating informal specifications into formal models using the topmost layer, known as the specifications layer. This layer corresponds to the CIM

layer in the MDA Y-chart. The resulting specification model, also known as MoT, establishes the scope of features (things), their properties and their relationship to the desired functionality.

Once the design specifications have been formalized, the subsequent step involves defining design microarchitecture through design model layer, which corresponds to MDA PIM layer. The MoT is transformed into the design model, known as MoD, via Template-of-Design (ToD). ToD is a domain-specific language (DSL) implemented in Python, which serves as a blueprint for constructing the MoD. The MoD itself is an instance of MetaRTL, a metamodel that encompasses various features of digital designs, such as ports and gates. Since the design model is independent of target language and technology semantics, the designer can focus solely on the design michroarchitecture, thus neglecting simulation semantics or synthesis artifacts [53]. Additionally, both generated APIs and manually extended APIs are available to facilitate a user-friendly design description.

The final layer, MoV, corresponds to PSM. MoV is the layer that is closest to the target code and is responsible for mapping the design model into different view models. This process enables the generation of the design in a preferred HDL with various coding styles. The currently supported HDLs are VHDL and Verilog, and the generated HDL code can be targeted for implementation on an ASIC or on an FPGA.

Relevance to this thesis

The RTL generation framework is extensively used to generate designs with fault injection capabilities effortlessly, thus enabling tool-agnostic fault simulation and emulation. Additionally, the existing framework is further enhanced to generate designs of mixed granularities. The detailed description is given in Chapter 5.

2.4.2 MDA Applied to Properties Generation and 4-eyes Principle

In addition to the RTL generation framework, MDA principles are adopted to generate structured formal verification properties. Devarajegowda et al. [48, 49] introduce a model-driven framework to generate properties called MetaProp that follows a three-layered structured approach, similar to the the RTL generation flow. The main purpose of Metaprop is to overcome major challenges for property generation such as informal specifications, grey-box approach and 4-eyes principle. Figure 2.6 provides a visual representation of the property generation flow (at the bottom of the figure), which complements the RTL generation flow (at the top of the figure).

The figure illustrates that both flows begin with the same set of formalized specifications model, MoT. Once MoT is established, a series of transformations take place independently in both flows. Template-of-Property (ToP) is a DSL (similar to ToD) coded in Python that transforms MoT into a less abstract model, namely Model-of-Properties (MoP). Additionally, ToPs are utilized to generate the property models for any supported micro-architecture.

The MoP is an instance of the MetaProp metamodel. This metamodel outlines how the property's structure is described in the MoP. The metamodel focuses on creating an abstract temporal trace, e.g., via temporal semantics, which captures the behavior of a sequential design over a specific time interval. With this concept in mind, the metamodel is defined to facilitate the creation and representation of properties within the formal verification framework [49].

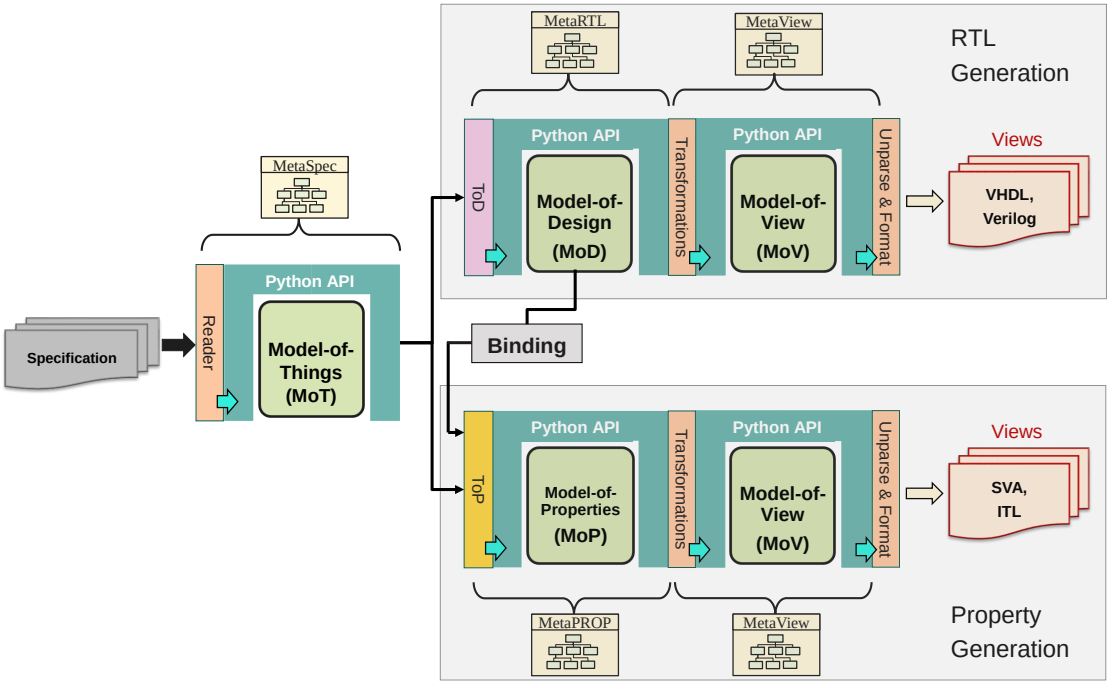


Figure 2.6: Property generation flow obeying 4-eyes principle

Lastly, the MoV layer generates various view models. The currently supported code languages are Interval Temporal Logic (ITL) and SystemVerilog Assertions (SVA). A comprehensive and detailed description of the property generation framework can be found at [47].

4-eyes principle

The adherence to the 4-eyes principle is a crucial requirement for both RTL and property generation flows. This principle serves to minimize the occurrence of errors and to prevent any bugs from going undetected due to inherent flaws in the generation flows. As an example, the verification step may not identify the bugs within the generation flow, since both the RTL and properties carry incorrect logic. To ensure its effectiveness, it is essential to separate the verification development process from the design implementation. To achieve this objective for both generation flows, a *Binding* mechanism has been developed, which allows for the provision of necessary design details specifically required for property automation. This mechanism ensures that the design verification flow remains isolated and separate from the intricate design implementation details [50]. The principle is also illustrated in Figure 2.6, where a clear distinction of property and generation flow is depicted.

Relevance to this thesis

Baudry et al. [28] states that: "If a fault occurs during a transformation process, it has the potential to introduce an error or fault in the resulting transformed model. If left undetected and unaddressed, this fault can propagate to subsequent models in the development process. As the fault continues to propagate, it becomes increasingly challenging to detect and isolate the source of the error". In this thesis, model transformation plays a significant role in incorporat-

ing fault injectors into the design model. Therefore, it is crucial to ensure the verification of the fault injectors' intended functionality. Additionally, it is important to verify that these transformations do not compromise the original functionality of the design when faults are disabled. To meet these requirements, the property generation framework offers a high level of flexibility while minimizing the need for manual intervention. Moreover, the framework is widely used to verify hardened processor cores. Further details regarding its application will be presented in the subsequent chapters.

Chapter 3

Functional Safety

A system is defined by Avizienis et al. [24] as "an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena". The system has a purpose and it must deliver its intended functionality without any deviation from the specifications. Naturally, a simple question arises: *How trustworthy is the system?* To answer this question, system engineers have come up with the concept of *dependability* that constitutes of the following attributes [24]:

- availability: "readiness for correct service ".
- reliability: "continuity of correct service" .
- safety: "absence of catastrophic consequences on the user(s) and the environment".
- integrity: "absence of improper system alterations".
- maintainability: "ability to undergo modifications and repairs".

As today's systems grow in size and complexity, their dependability is becoming more vulnerable to threat factors such as faults, failures and errors. A fault is an abnormal physical condition that occurs on the system. An error is a manifestation of a fault and causes a deviation of the system from the expected behavior. Lastly, a failure is the observable deviation caused by the error(s). These threats can have catastrophic consequences such as endangering human life, thus rigorous and stringent verification and validation techniques are required to asses the system dependability. *Time to Failure* (TTF) is one of the most important metrics of dependability which evaluates the period of time between system's operation start and the occurrence of a failure, e.g., if a failure is observed six months after the system started operating then the TTF is six months. Mean Time to Failure (MTTF) is calculated by averaging the time between multiple failures in a device while *Failure in Time* (FIT) represents a failure in one billion (10^9) hours [11]. The FIT rate of a system having n components is calculated as [11]:

$$FITrate_{system} = \sum_{i=0}^{n} FITrate_{component_i} \tag{3.1}$$

This thesis focuses on evaluating the threat factors on *safety* and *reliability* attributes of the system's dependability.

Definition 1 [Reliability]:
Reliability $R(t)$ is defined as the probability that no failures occur on a system during time interval $(0,t]$, i.e., the probability that the system is still functioning without any failures at time t [11]. Assuming a set of N_s similar devices, $R(t)$ is the subset of devices N_c from N_s that do

not experience failures after time t. Mathematically this relation can be expressed as following when considering $N_f(t)$ as the subset that encounter failures during $(0,t]$ [11]:

$$R(t) = \frac{N_c}{N_s} = \frac{N_s - N_f(t)}{N_s} = 1 - \frac{N_f(t)}{N_s} = 1 - P(Failure) \tag{3.2}$$

Definition 2 [Safety]:
Safety is defined as the probability that a system functions in a correct manner or fails in a "safe" mode [11]. The main difference between safety and reliability is the "fail-stop" and "fail-fix" behavior of safe systems, i.e., the safe system will either fix the fault or stop operating after a fault occurs such that its effects are avoided or preventing further damages.

In this chapter, the main concepts regarding faults are outlined. These fundamental concepts contribute to the most common safety verification technique such as *fault injection*. Next, the prevalent automotive safety standards are described. The chapter concludes with safety design and verification techniques adhering to these standards.

3.1 Fault Concepts

Around two decades ago, the scaling of technology nodes approximating their limits propelled dependability to gain a lot of traction as a major design challenge by multiple researchers [71]. The continuous scaling facilitates various factors to compromise design's reliability by introducing faults. Faults are categorized into two main classes: *systematic faults* and *random faults*.

Systematic faults are caused by human error during the design, manufacturing or verification processes. These kind of faults manifest as design flaws, process variations and verification gaps. Systematic faults are consistent and predictable, therefore a thorough, rigid and robust design/verification process can mitigate the fault effects. The 4-eyes principle (described in Chapter 2) should be employed to identify and address systematic faults effectively.

Random faults are typically caused by external factors and occur sporadically in the design. These kind of faults appear randomly and unpredictably at different parts of designs that can lead to serious failures in the design behavior. Various design and verification practices are required to ensure the reliability and safety of the design in the presence of random faults, such as fault correction/detection and fault injection.

Due to the above reasons, it is necessary to develop various concepts regarding faults in order to abstract the effect of the physical failures (faults) and at the same time to ease the verification process. In the following, a brief overview is given on different fault concepts such as *fault modeling*, *fault testability* and fault *fault collapsing*.

3.1.1 Fault Modeling

A *fault model* represents the physical fault on a particular abstraction level in order to reduce the complexity for analysis purposes. This thesis focuses only on applicable fault models on the circuit level (RTL and/or gate-level) and omits the models on transistor level or higher abstraction levels. The dominant fault models are categorized as *permanent*, *intermittent*, and *transient*.

Permanent faults

Permanent faults (hard errors) persist on the design for the full duration of its operation and manifest as *stuck-at*, *bridging*, or timing faults. Stuck-at faults tie the signal's value to a permanent logic value: stuck-at-1 (short circuit) and stuck-at-0 (open circuit) while bridging faults represent circuit shorts between two or more nearby wires in a region of the design. Meanwhile, timing faults manifest as a delay of the correct calculation of output values. Permanent faults emerge as a result of manufacturing variations [71], electro-migration [71], temperature [71], aging [71], processing defects, and others. Moreover, even a small magnitude of Random Dopant Flunctuations can largely affect the behavior of the ever-shrinking MOSFET transistor cells [22], [71].

Intermittent faults

Intermittent faults occur occasionally on unstable designs due to process variations and manufacturing residuals, and, often precede permanent faults [43]. These kind of faults appear repeatedly at the same location [43], e.g., an occasionally short circuit. Often, the faults occur in bursts during irregular intervals [43].

Transient faults

Transient faults (soft errors), commonly known as Single Event Effects (SEEs), appear only for a temporary period and manifest as a bit-flip in a logic gate or a memory cell. "Left unchallenged, soft errors have the potential for inducing the highest failure rate of all other reliability mechanisms combined" as stated by Baumann et al. [29]. Nevertheless, combinational cells are very resilient to these kind of faults [99] due to masking effects. By far the highest impact is observed in memories/flip-flops. Alpha particles (radioactive decay in the device package) [29] and/or cosmic rays [29], [122] can create an electron-hole pair in the bulk substrate of the MOSFET, thus temporarily reversing its logic value. In contrast to permanent faults, soft errors are fixable, e.g., by re-writing the memory cell.

3.1.2 Fault Testability

A fault does not necessarily mean that it will have detrimental consequences on a design's behavior. The most common technique to measure the fault impact is *fault testing*, i.e., applying appropriate input test sequences to measure the effect of a specific fault into the Primary Outputs (POs) of the design. In the following, the fundamental concepts regarding testing [14] are described:

- Fault Sensitization : a fault location is sensitized if the test sequence alters the logical value of the particular location in the presence of the fault, e.g., a stuck-at-1 fault is sensitized when the test drives a logical '0' to the particular location.
- Fault Propagation: occurs when the fault affects further locations other than the initial one, i.e., propagates through a design's path.
- Fault Observability: if the fault effect has propagated to the POs of the design, then the fault is deemed to be observable.

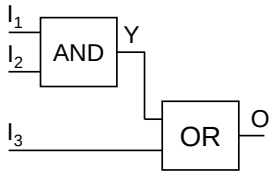
- Fault Controllability: ability to set certain logical values at design's signals, i.e., making a fault observable by setting appropriate input stimuli.
- Fault Testability: the capability to control and observe the fault through a suitable test pattern.

As described above, adequate test patterns are a prerequisite to make a fault observable. ATPG has become the prevalent *structural test* methodology to test the correctness of digital designs. ATPGs try to generate compact test patterns that are able to detect all faults, but considering the design complexity there are cases where the ATPG fails to find a proper pattern due to long runtimes (NP-complete problem) or design redundancy. Deriving from the forementioned problem, *functional tests* are utilized to test large-sized processor-subsystem designs where a structural test is not feasible. A functional test is represented by a piece of software code embedded into the memories that attempts to sensitize as many faults as possible.

3.1.3 Fault Collapsing

In theory, for m applicable fault models and for l possible fault locations, the total fault set is $m \times l$. The *fault collapsing* technique decreases the total number of faults in a digital design to speed-up and improve the fault testing methodologies. Structural fault collapsing considers only the topology of the design while functional fault collapsing utilizes design's behavior properties [127]. The later considers the behavior of the design in the presence of a fault, i.e., two faults are equivalent if they produce the exact same functional behavior, e.g., identical POs in presence of the faults.

Structurally, fault collapsing is achieved by exploiting the concepts of *fault equivalence* and *fault dominance*. A fault F_i dominates another fault F_j when all the test patterns that detect F_j will also detect F_i. If a fault dominates another fault, then the faults are called equivalent, thus, only one of them needs to be considered during test pattern generation [127].



Inputs			Output										
I_1	I_2	I_3	Fault-free	I_1 / sa-0	I_1 / sa-1	I_2 / sa-0	I_2 / sa-1	I_3 / sa-0	I_3 / sa-1	Y / sa-0	Y / sa1	O / sa-0	O / sa-1
0	0	0	0	0	0	0	0	0	1	0	1	0	1
0	0	1	1	1	1	1	1	0	1	1	1	0	1
0	1	0	0	0	1	0	0	0	1	0	1	0	1
0	1	1	1	1	1	1	1	0	1	1	1	0	1
1	0	0	0	0	0	0	1	0	1	0	1	0	1
1	0	1	1	1	1	1	1	0	1	1	1	0	1
1	1	0	1	0	1	0	0	1	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	0	1

Figure 3.1: Example of a fault matrix

As an example, let us assume the circuit shown in the left side of Figure 3.1. There are

in total five fault locations (I_1, I_2, I_3, Y, O) and ten different faults when stuck-at (sa) fault model is taken into account. The truth table of the circuit is shown in the right side of Figure 3.1 while fault models are considered. It is clear that that two sets of faults are functionally equivalent such as I_1/sa-0, I_2/sa-0, Y/sa-0 (marked with red lines) as well as I_2/sa-1, Y/sa-1, O/sa-1 (marked with blue lines). Therefore, only one fault per set should be considered (e.g., Y/sa-0 and O/sa-1) instead of three and hence the total fault set is reduced into only six faults.

3.2 Automotive Safety Standard

Electronic designs constantly interact with humans in their daily life. The random nature of the faults in digital designs can have detrimental consequences for humans, e.g., financial losses, security breaches, or even life-threatening risks. To deal with the unpredictable and undesirable failures, it is essential to have a standardization across different domains of safety-critical systems. Standards create a common uniform, stable and reliable framework/environment that all its users need to adhere to. **IEC 61508** [2] is the international functional safety standard of electronic designs that defines the overall requirements, specifications, rules and constrains of safety-critical designs, their verification and validation. One of the notable features of IEC 61508 is its *safety lifecycle* approach, which outlines safety management stages throughout the system's lifespan. Safety lifecycle is categorized into five primary stages [3]: *(i) Overall safety scope definition, (ii) Hazard and risk analysis, (iii) Planning and realisation, (iv) Operation, maintenance and repair, (v) Decommissioning or disposal*. The automotive industry have adopted IEC 61508 including its principles and created the automotive safety standard **ISO26262** [12]. ISO26262 defines functional safety as "absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical/electronic systems" [12]. In this thesis, ISO26262 serves as a guiding directive for the developed methodologies and a brief description is given next.

3.2.1 Automotive Safety Integrity Level

Automotive Safety Integrity Level (ASIL) is a risk classification system from ISO26262 standard. ASIL uses a four-level categorization system denoting each level with letters A, B, C, and D, where A represents the lowest risk and D the highest risk. As an example, the breaking system of the car would be classified as highest risk ASIL-D, while the the sound system would be classified as low-risk ASIL-A. The classification is determined by the process of Hazard Analysis and Risk Assessment (HARA). This process identifies and categorizes all hazards and dangerous events that could undermine the required *safety goals*. The safety goals represent the necessary requirements that an automotive item/component must meet to operate safely and without endangering the vehicle.

HARA identifies and determines the hazards using the impact factors such as *severity*, *probability of exposure* and *controllability during a failure* which are shown in Figure 3.2. Each impact factor is distinguished in different classes, i.e., S0-S4 for severity, E0-E4 for probability of exposure and C0-C4 for controllability in case of an failure. An event with the lowest severity level S0 could be an light incident, e.g., a car scratch while an accident with another card would be classified as S3. An E0 probability represents a highly unlikely event and an E4 probability

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

Figure 3.2: HARA impact factors classes [12]

represents the most probable event, e.g car breaking. An example for high controllability C0 could be the fuel level and a low controllability C3 could be failure of the airbag.

After HARA categorization, ASILs are determined by combining all the impact factors together as shown in Figure 3.3. As can be seen from the matrix illustrated in the figure, ASIL classification (A, B, C, D) is done according to the combination of impact factors. A component that in presence of a hazardous event has a severity S3, an exposure probability E4 and a controllability C3 is classified as an ASIL D component. Additionally, some configurations of severity, probability and controllability are attributed to quality management (QM) due to not posing a threat to the automotive vehicle.

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

Figure 3.3: ASIL determination [12]

3.2.2 Fault Classification

A fault can affect a design behaviour in different ways. ISO26262 presents a classification mechanism for failures, which is determined by analyzing the causes of failures and the system's capacity to detect and correct them. A complete overview of the classification flow is

given in Figure 3.4. The flow considers the effects of the faults on safety-critical elements and whether they can violate the predefined safety goals during the risk assessment phase.

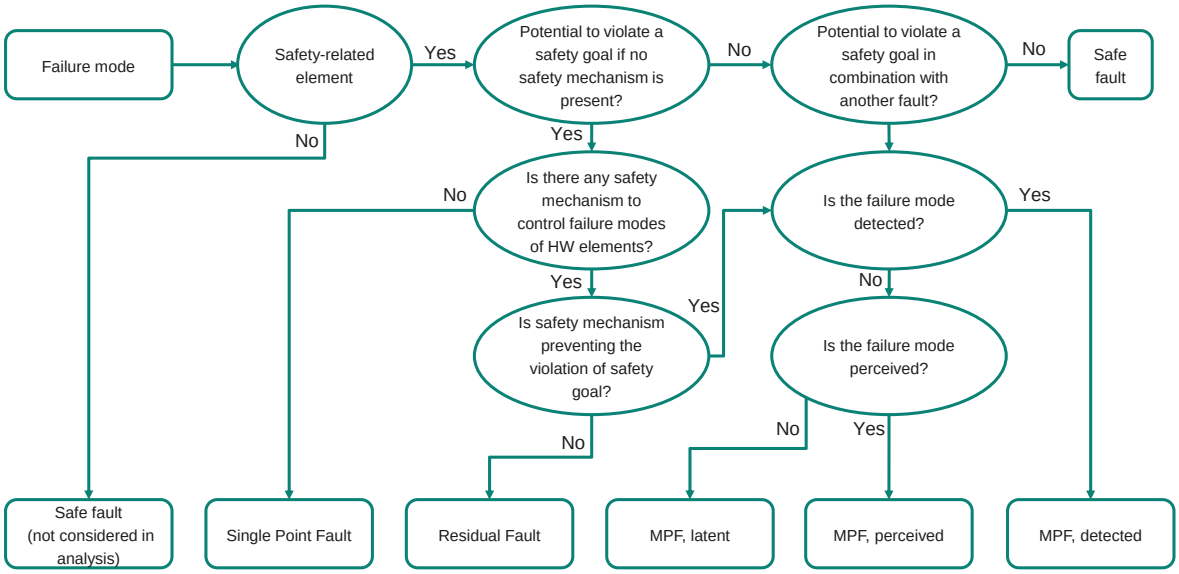


Figure 3.4: Fault classification system [12]

The faults are classified into six different categories as following:

- **Safe faults (SF)**: a fault is considered safe if it is not related to any safety-critical hardware element or if it does not violate a safety goal of safety-critical elements even in combination with other independent faults due to masking effects.
- **Single-point faults (SPF)**: a single point fault affects safety-critical elements and there is no safety mechanism to either detect or correct them.
- **Residual faults (RF)**: Residual faults have a similar impact on safety-critical elements as single-point faults. However, they specifically affect areas that should be protected by safety mechanisms. Unfortunately, in these cases, the safety mechanisms fail to safeguard the affected area, leading to a violation of the safety goals.
- **Detected multi-point fault (DMPF)**: detected MPF are multiple independent faults that are detected or corrected by a safety mechanisms in a safety-critical element.
- **Perceived multi-point fault (PMPF)**: these kind of multiple independent faults are neither corrected or detected by the safety mechanism, but they are perceived by the automotive user because they have an impact on the car behaviour, e.g., the driver might observe a problem with the lights even though there is no indication by the system.
- **Latent multi-point fault (LMPF)**: a multiple point fault that is neither perceived nor detected is considered a latent MPF.

3.2.3 Hardware Fault Coverage Metrics

ISO26262 introduces three different metrics regarding fault coverage such as Probability Metric for random Hardware Failures (PMHF), Single-Point Fault Metric (SPFM), and Latent Fault Metric (LFM). SPFM represents the robustness against SPF and RF while reflects the robustness against latent faults. These metrics can be derived from the qualitative safety analyses

by incorporating numerical data and enhancing the analysis. The standard provides means to calculate the metrics as follows [12]:

$$PMHF = \lambda_{SPF} + \lambda_{RF} + \lambda_{MPFLatent} \tag{3.3}$$

$$SPFM = 1 - \frac{\sum \lambda_{SPF} + \lambda_{RF}}{\sum \lambda} \tag{3.4}$$

$$LFM = 1 - \frac{\sum \lambda_{MPFLatent}}{\sum \lambda - \lambda_{SPF} - \lambda_{RF}} \tag{3.5}$$

where:

λ	=	total failure rate of safety-related design elements
λ_{SPF}	=	failure rate of SPF
λ_{RF}	=	failure rate of RF
$\lambda_{MPFLatent}$	=	failure rate of latent MPF

These three key metrics are required for ASIL classification as illustrated in Table 3.1. For example, as can be seen from the table, a classified ASIL-D component requires: *(i) a probabilistic risk quantification less than 10 FIT rate, (ii) greater than 99% fault coverage (robustness) against SPF, and (iii) greater than 90% coverage against LMPF*. Every automotive product that requires ISO 26262 certification must comply with the requirements outlined in the table.

Table 3.1: Fault coverage metrics for ASIL classification [12]

	ASIL A	ASIL B	ASIL C	ASIL D
PMHF	<1000 FIT	<100 FIT	<100 FIT	<10 FIT
SPFM	-	$\geq 90\%$	$\geq 97\%$	$\geq 99\%$
LFM	-	$\geq 60\%$	$\geq 80\%$	$\geq 90\%$

3.3 Standard-Compliant Safety Design

To mitigate the adversary and severe effects of random faults, robust design systems are required by ISO26262 standard. Design robustness can be achieved through *fault avoidance* and *fault tolerance*. As the term suggests, fault avoidance aims to prevent or reduce the likelihood of faults occurrence by a careful and rigid design and verification process. In order to achieve the required fault avoidance, engineers can deploy diverse design systems followed by a structured verification process through stringent design/verification rules. Nevertheless, the random and unpredictable nature of faults makes it unfeasible to ensure a 100% fault avoidance. Typically, fault avoidance is often used in combination with fault tolerance to achieve a high level of system reliability. Fault tolerance focuses and emphasizes detection, mitigation and recovery from faults occurrence, Commonly, tolerance is achieved through *design hardening*.

Definition 3 [Design hardening]:
Design hardening is the process of protecting a digital design against unexpected behavior in the presence of physical faults by correcting and/or detecting their effect.

There exist different hardware-based safety mechanisms and techniques that enable design hardening through *information redundancy*, e.g., Error Detection and Correction Codes (ECC) and through *spatial redundancy* approaches. System requirements, e.g., area and fault tolerance, and performance constraints determine which is the best mechanism to use for specific use cases. In the following, a brief overview of the common safety mechanisms is given.

3.3.1 Safety Mechanisms based on Information Redundancy

Error Correction Codes

ECCs are safety mechanisms widely utilized to protect memory elements against single and multiple bit errors. ECCs implement algorithms that can detect and/or correct differences between a *received* signal value and its *expected* value. The main feature of ECC designs is adding extra redundant bits to the original signal (data). The simplest error detection mechanisms consist of *parity codes*, i.e., memory elements are extended by an extra parity bit detecting single errors. The parity bit is relatively simple to calculate with low overhead, but these kind of codes can only detect errors.

Typically in complex and safety-critical designs, an ECC consists of an encoder and a decoder. Encoder contains mathematical coding algorithms, e.g., Convolution Codes and Linear Block Codes, to calculate the redundant bits. When the data is received/read, the decoder applies similar algorithms to check whether the data is corrupted due to the errors. Well-known ECCs comprise Hamming codes, Reed-Solomon codes, Bose-Chaudhuri-Hocquenghem (BCH) codes and Low-Density Parity-Check (LDPC) codes. Various ECC schemes exist and they are classified by the number of errors they can detect/correct such as Single Error Correction (SEC), Single Error Detection (SED), Single Error Correction and Detection (SEC-DED), Double Error Correction and Triple Error Detection (DEC-TED) etc..

Figure 3.5 shows a SEC-DED Hamming code based ECC that protects 16-bit data written to a register. The ECC encoder uses the Hamming algorithm to calculate the check bits width. The width can be obtained by solving the following *Hamming Bound* inequation [38]:

$$\frac{2^q}{\sum_{k=0}^{\frac{d-1}{2}} \frac{q!}{k!(q-k)!}} \geq 2^w \tag{3.6}$$

where w is the encoder input width, q is encoder output width (cw_in) and d is the *Hamming distance*. Hamming distance is a metric that specifies how many bits can be corrected and detected, and for a SEC-DED algorithm, the distance is 3. By solving Equation 3.6 for d=3, output width is equal to 21. As a result, only 5 check bits (cb) are required since check bits width is calculated by $n-k$.

The register stores 21 bit input data that is read by the decoder. The decoder calculates an expected value and compares it with the received data value. If only a single bit error is detected, the decoder is able to correct it and raise the "err_1" flag. When 2 bit errors are detected, the "err_2" flag is raised but the data remains corrupted. In general, most of ECC blocks follow the same principles as the explained SEC-DED and more information can be found on the available literature.

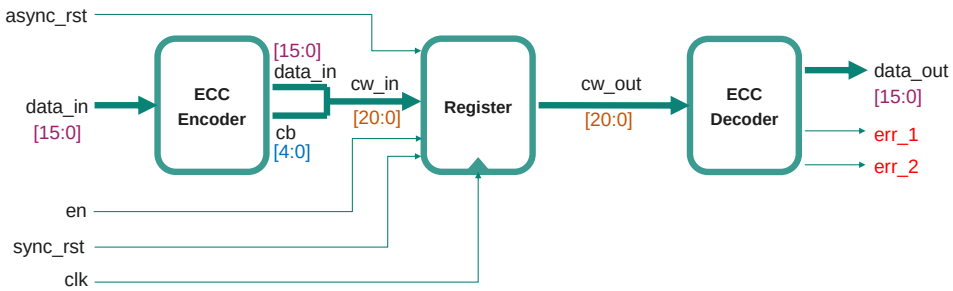


Figure 3.5: SEC-DED ECC block

Cyclic Redundancy Check

Cyclic Redundancy Check (CRC) codes are error detection codes mostly used in memory or data transmission systems. Similarly to ECCs, CRC adds redundant bit, also known as *checksum*, to the original input data via an encoder. CRC requires a fixed generator polynomial $P(x)$ and a polynomial division is performed between $P(x)$ (divisor) and the input data (divident). The remainder of division, i.e., checksum is appended to the original input data. On the decoder side, the same polynomial division is performed using an identical $P(x)$. The new calculated checksum from the decoder is compared with the received checksum and if any mismatch is detected, an error is detected. In general, CRC codes have a lower overhead compared to ECCs but offer only error detection without correction.

3.3.2 Safety Mechanisms based on Spatial Redundancy

Spatial redundant safety mechanisms are a classical approach to increase fault tolerance by duplicating or triplicating the critical parts of the design. Commonly, redundant systems are called *lockstep* systems. Due to their fault-tolerance benefits, Lockstep is widely utilized in many industrial safety-critical designs such as AURIX ™32-bit microcontrollers [1].

The typical implementations of Lockstep are Double Module Redundancy (DMR) and Triple Module Redundancy (TMR). All the mechanisms work similarly, i.e., identical redundant modules run in parallel and their outputs are compared. Nevertheless, there are differences in terms or area and fault tolerance capacity, thus system trade-offs are required to fulfill requirement constraints. Figure 3.6 represents a high level block description of DMR (left side) and TMR(right side). As can be seen from the figure, the main difference between them is the number of redundant modules, i.e., TMR consists of three identical modules while DMR requires only two modules, thus TMR requires more area.

Let us consider DMR functionality first. Both the identical modules are running in parallel and same inputs are fed to them ensuring that the same data is being processed. The system outputs are connected to one of the modules output lines while the *comparator* compares the outputs of the modules to check for their consistency. If there is a mismatch, an error flag is raised indicating one or more potential fault(s) in one of the modules. As a result, DMR offers error detection capabilities but no error correction. On the other side, TMR enables both error detection and correction. A *voter* mechanism compares the outputs of the three modules and if the outputs differ, the error flag is raised. Additionally, the voter calculates the majority output, i.e., the output that is identical by two out of three modules. The system output is connected to

voter's calculated output, resulting in an corrected error. However, if two modules are faulty, TMR would proceed with a faulty output.

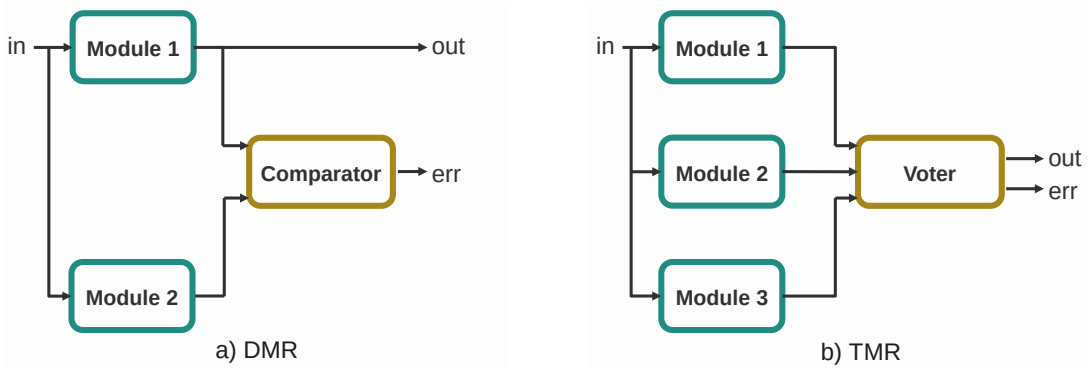


Figure 3.6: DMR and TMR

A comprehensive description of various fault tolerant architectures is given by Sorin [123].

3.4 Standard-Compliant Safety Verification

Safety mechanisms provide means to protect against random and hazardous faults, nevertheless, stringent verification and validation techniques are required to evaluate the system's robustness and reliability. ISO26262 recommends *fault injection* to verify system's dependability with a high degree of confidence. Fault injection is defined as *the deliberate introduction of controlled faults into the system and by observing its behavior in the presence of injected faults* [20]. The main goal is to asses the system's ability to detect, correct and recover from faults during controlled experiments. Fault injection methods utilize various predefined fault models during the experiments, e.g., permanent or transient faults. The fault injection experiments include three major steps: (i) injection according to desired fault models, (ii) monitoring and analysis, (iii) evaluation of the results. Each of these steps should adhere to ISO26262 requirements to achieve the intended certification, e.g., desired ASIL. This section gives a general overview of the fault injection process, fault injection attributes and a description of the available fault injection techniques and tools.

3.4.1 Overview of the Fault Injection Process

Figure 3.7 illustrates a complete overview of the fault injection process. The design that is a subject to fault injection is depicted as DUT. Similarly to the traditional verification methods, a *stimulus* is required to evaluate design's behavior and functionality. The main function of the stimulus is to exercise the DUT under various scenarios. This could be achieved through functional tests or ATPG tests (as mentioned earlier in the chapter). The *fault list*, also known in literature as fault library, contains all fault injection attributes such as fault type, fault location, and fault injection time. The user can define the fault list by referring to DUT information, e.g., design signals' names. The *controller* simply regulates and controls the fault injection process and provides the fault list information to the *fault injector*. The fault injector drives and injects various faults into the DUT complying with the fault list description. The *monitor* observes

design's outputs and internal states during the process and the *analyzer* classifies and categorizes the faults according to their effect in the design, e.g., safe fault or a failure. Typically, the analysis is done by comparing the values of outputs and internal registers in presence of faults with their expected non-faulty values.

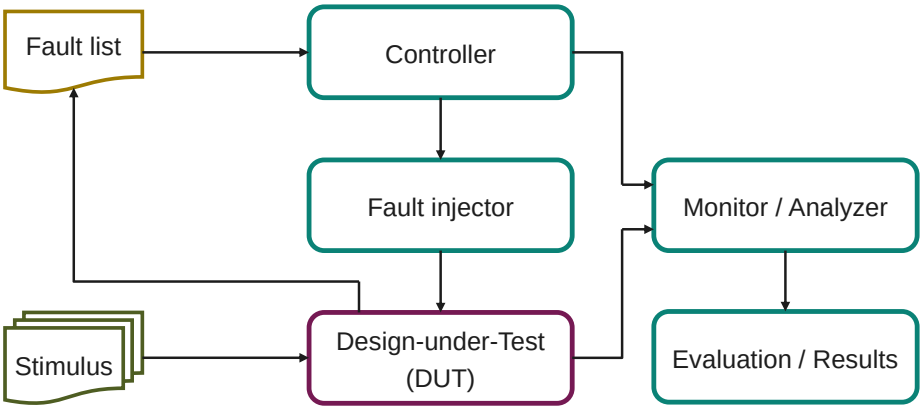


Figure 3.7: Fault injection process

The results of a fault injection process are logged into files containing information about their effect. In the end, the safety engineer evaluates the design's susceptibility to various fault scenarios and pinpoints opportunities for enhancing its fault tolerance mechanisms. During the fault injection process there are many factors to consider which will be explained in the following.

3.4.2 Fault Injection Attributes

Fault injection is a prerequisite to evaluate the robustness of a system. Major factors to be considered during different fault injection campaigns are fault injection attributes. Fault injection attributes encompass distinct characteristics or properties of injected faults during fault injection experiments. They dictate the nature and behavior of these faults, playing an important role. The most common attributes are: (i) *fault type*, (ii) *fault location*, (iii) *fault injection time*, and (iv) *fault multiplicity*.

Definition 4 [Fault location]:
A fault location specifies the exact signal in the design, i.e., location in the system where the fault should be injected, e.g., registers, memories, etc.. The set of all fault locations of the design is represented by the set $F_L = \{s_0, s_1, ..., s_{n-1}\}$ that denotes all signals of the design that can be an object of fault location, where n represents the total number of signals. The single fault location F_{L_i} is an element of F_L where i denotes the index of element such that $0 \leq i < n$.

Definition 5 [Fault type]:
The fault type specifies the fault model type to be injected. The temporal fault models used in this thesis are represented by two major distinct sets:

- permanent faults: $P_F = \{stuck\text{-}at\text{-}1, stuck\text{-}at\text{-}0, delay faults\}$
- transient faults: $T_F = \{bit\text{-}flip\}$

The set of all fault types is the set $F_{TP} = P_F \cup T_F$ and the particular fault type to be injected is denoted by F_{TP_i} such that $F_{TP_i} \in F_{TP}$.

Definition 6 [Fault injection and release time]:
Fault injection time represents the exact time at which the fault is injected during the design's operation. Let T be the time interval during which the design is operating. Fault injection time $F_{TM}(t_i)$ represents the time t_i during interval T when a fault is introduced in the design and $F_{TM}(t_r)$ represents the time t_r during interval T when a fault is removed from the design.

According to [32, 104], fault-injection methods require only these three essential attributes, i.e., fault type, fault location and fault injection/release time to effectively inject faults into a design model. The *fault space* ξ is a three-dimensional space whose dimensions include the exact time of occurrence and duration of the fault (when), the type or form of faults (how), and the location of the fault within the design (where) [32], i.e., the fault space covers the combination of the aforementioned fault injection attributes. Additionally, considering the complexity of designs nowadays and strict ISO26262 requirements, another important attribute of fault injection is fault multiplicity.

Definition 7 [Fault multiplicity]:
Fault multiplicity refers to the number of faults to consider during fault injection campaigns. Typically, fault multiplicity is categorized into *single-fault* injection and *multiple-faults* injection. The single-fault (SF) injection represents a single fault space $SF = \xi$, i.e., involves injecting only one fault at a time into a location of the DUT. SF are important for the proper evaluation of ISO26262 metric such as SFM. A multiple-fault (MF) injection considers a set of different fault spaces $MF = \{\xi_0, \xi_1, ..., \xi_{M-1}\}$, where M represents the number of faults to inject. In a multiple fault scenario, multiple faults are injected into the DUT either simultaneously or sequentially, thus enabling LFM evaluation. Single fault analysis allows for the isolated study of individual faults, uncovering their specific impacts. Conversely, multiple fault analysis offers a holistic view, assessing the system's overall resilience during real-world fault scenarios.

3.4.3 Fault Injection Techniques

Fault injection is a versatile process capable of targeting different system levels, addressing both hardware and software faults. This adaptability has stimulated the development of numerous fault injection techniques within academia and industry. Most common techniques can be categorized as : (i) *hardware-based fault injection*, (ii) *software-based fault injection*, (iii) *simulation-based fault injection*, and (iv) *emulation-based fault injection*. Each of these technique possesses unique characteristics and advantages, making them suitable for specific testing scenarios aimed at evaluating distinct features of system reliability and fault tolerance. This section provides in the following a short overview of common fault injection techniques, highlighting their distinctive features, advantages, and limitations.

Hardware-Based Fault Injection

Hardware-based fault injection utilizes specialized test hardware tools to inject faults at a physical level of the DUT. These tools disturb the hardware by subjecting it to various environmental parameters such as heavy ion radiation, electromagnetic interferences, etc. This can be achieved by injecting voltage sags on the power rails (power supply disturbances), performing laser fault injection, or altering the values of the circuit's pins [136]. Hardware fault injections are performed on actual circuit after fabrication [32], subjecting the circuit to interference to

produce faults. Both permanent and transient faults can be injected. Such tests, although time-consuming, are faster than simulations and are often conducted just before or during production to assess circuit reliability [32]. Hardware-implemented fault injection methods can be classified into two categories based on the types of faults and their locations [136]: (i) *hardware fault injection with contact* where the injector has a direct contact with the circuit pins, and (ii) *hardware fault injection without contact* where the injector has no direct contact the target circuit.

Software-Based Fault Injection

Software-based fault injection includes injecting faults at the software level, typically in the form of software errors by modifying the original source code to introduce faults and alter the system's state, i.e., to mimic the occurrence of hardware faults. Various faults can be injected, including register and memory faults, dropped or replicated network packets, and erroneous error conditions [32, 136]. Software fault injections focus on implementation details, addressing program states, communication, and interactions. Simulations with faults take longer due to capturing system operations and timing aspects accurately [32]. Software fault injections can be non-intrusive, but timing considerations may cause disruptions. The injection mechanism running on the same system as the software being tested can influence timing results [136].

Simulation-Based Fault Injection

Simulation-based fault injection involves modeling and simulating both the target system and potential hardware faults. The simulation process modifies either the hardware model or the software state of the target system, making it behave as if there were a hardware fault [89, 86]. Simulation-based fault injection techniques can be classified into two categories [89]: (i) *run-time fault injection*, and (ii) *compile-time fault injection*. Runtime fault injection involves injecting faults during the simulation or execution of the model, while compile-time fault injection introduces faults at compile-time in the target hardware model or software executed by the target system [89]. Simulation-based fault injection offers several advantages such as there is no risk of damaging the system under normal operation. Additionally, these techniques are more cost-effective compared to hardware-based methods [89].

Fault simulation can be applied by: (i) modifying the circuit HDL description by adding *saboteurs* and/or *mutants* or (ii) utilizing a special software tool typically known as fault simulator. A saboteur is an additional component incorporated into the hardware design at a specific location for fault injection purposes. Once activated, it modifies the value of one or more signals, thereby introducing the desired fault [32]. A mutant is a specialized model that includes inactive code blocks within the regular circuit description. By injecting faults, these code blocks can be activated, thereby altering the behavior of the logic device. Since the fault response originates internally within the model, fault injection can be performed at various levels of abstraction [136]. On the other hand, non-modifying fault simulator tools depend largely on the application, availability and built-in commands of used tools.

Emulation-Based Fault Injection

Emulation-based fault injection techniques utilize emulation platforms, e.g., hardware prototyping on FPGA-based logic emulation systems. In the context of fault injection, fault emulation has been proposed to overcome time limitations imposed by fault simulations [136]. This approach involves implementing the circuit on an FPGA using classical synthesis and routing design flow. A development board connected to a host computer allows for defining fault injection campaigns, controlling experiments, and displaying results [136]. To inject faults, modifications may be required in the circuit description to remain synthesizable and adhere to emulator hardware constraints. However, generating instrumented circuit descriptions for fault injection can be time-consuming due to multiple reconfigurations. Another approach, called run-time reconfiguration emulation-based fault injection, avoids instrumenting the circuit description by relying on FPGA's built-in reconfiguration capabilities [18].

Comparison of different techniques

Table 3.2 and Table 3.3 display distinctions of the fault injection techniques highlighting their characteristics, advantages and disadvantages.

Table 3.2: Characteristics of fault injection techniques [72]

Characteristics	HW-based + contact	HW-based - contact	SW-based	Simulation -based	Emulation -based
Fault injection points	Limited set of injection	Internal (soft errors)	Only locations accessible to SW	Full access to HW blocks	Full access to HW blocks
Able to model permanent faults	Yes	No	No	Yes	Yes
Genericity	Medium	Medium	Low	High	High
Intrusiveness of the experiment	None	None	High	None	None
Observability	Low	Low	Low	High	High
Controllability	High	Medium	High	High	High
Repeatability	Medium	Medium	Medium	High	High
Automatization	Medium	Medium	High	High	Medium
Precision	High	High	Medium	Low	Low
Experiment speed	Real time	Real time	Real time	Very slow	Faster than sim.
Cost	High	High	Low	Medium	Medium

Table 3.3: Summary of advantages and disadvantages of fault injection techniques [136]

	Advantages	Disadvantages
Hardware-based	• Can access locations that is hard to be accessed by other means. • High time-resolution for hardware triggering and monitoring. • Well suited for the low-level fault models. • Not intrusive. • Experiments are fast. • No model development or validation required. • Able to model permanent faults at the pin level.	• Can introduce high risk of damage for the injected system. • High level of device integration, multiple-chip hybrid circuit, and dense packaging technologies limit accessibility to injection. • Low portability and observability. • Limited set of injection points and limited set of injectable faults. • Requires special-purpose hardware in order to perform the fault injection experiments.
Software-based	• Can be targeted to applications and operating systems. • Experiments can be run in near real-time. • Does not require any special-purpose hardware; low complexity, low development and low implementation cost. • No model development or validation required. • Can be expanded for new classes of faults.	• Limited set of injection instants. • It cannot inject faults into locations that are inaccessible to software. • Does require a modification of the source code to support the fault injection. • Limited observability and controllability. • Very difficult to model permanent faults.
Simulation-based	• Can support all system abstraction levels. • Not intrusive. • Full control of both fault models and injection mechanisms. • Low cost computer automation; does not require any special-purpose hardware. • Maximum amount of observability and controllability. • Allows performing reliability assessment at different stages in the design process. • Able to model both transient and permanent faults	• Large development efforts. • Time consuming (experiment length). • Model is not readily available. • Accuracy of the results depends on the goodness of the model used. • No real time faults injection possible in a prototype. • Model may not include any of the design faults that may be present in the real hardware.
Emulation-based	• Injection time is more quickly compared with simulation-based techniques. • The experimentation time can be reduced by implementing partially or totally the input pattern generation in the FPGA. These patterns are already known when the circuit to analyze is synthesized.	• The initial VHDL/Verilog description must be synthesizable and optimized to avoid requiring a too large and costly emulator and to reduce the total running time during the injection campaign. • The cost of a general hardware emulation system and/or the implementation complexity of a dedicated FPGA based emulation board. • The emulation is only used to analyze the functional consequences of a fault. • When using an FPGA-based development board, the main limitation becomes the number of I/Os of the programmable hardware. • Necessity of high speed communication link between the host computer and the emulation board.

Relevance to this thesis

This section provides an overview of the automotive safety standard ISO26262 and its focus on dependability aspects, which include key fault concepts. Furthermore, it presents a visualization of safety design and verification techniques compliant with the standard, along with brief illustrative examples. This thesis extends the existing state of the art fault handling and analysis by providing a low-effort automatic model-driven approach to handle and analyze different fault models. Additionally, novel simulation-based and emulation-based fault injection techniques have been developed that overcome challenges of state of the art techniques. Details will be given in the following chapters.

3.4.4 Formal Verification

ISO26262 recommends formal and semi-formal methods to verify safety critical components to achieve the required ASIL level. Formal verification is the common formal method for addressing intricate safety-verification challenges. Formal verification is defined as the application of precise mathematical proof techniques to validate the properties of a design implementation. This approach treats the Design Under Verification (DUV) as a mathematical model, rendering it applicable to mathematical proof methodologies. The common characteristic of formal verification is exhaustiveness, implying the requirement to examine all possible input-state combinations that impact a specific aspect of the design, often referred to as the Cone of Influence (COI). A formal verification approach comprises three primary components: *Proof Methods* (e.g., SAT-solving, Graph representations of Boolean functions, Finite state machine traversal), *Languages* (e.g., propositional logic, temporal logic, predicate logic), and *Modeling* (e.g., Boolean network, Finite state machine). These components significantly influence the proof duration [93]. Formal verification serves the dual purpose of verifying implementation through *Equivalence Checking* and verifying design's behavior via *Property Checking*, including techniques such as *Model Checking*.

Model Checking

Model Checking, in essence, involves evaluating whether a finite-state model representing a system adheres to its specified behaviors using temporal logic. It accomplishes this by examining all possible future states to determine if any property violations occur [42]. To facilitate this analysis, a model checker translates properties, often expressed in conventional syntax like System Verilog Assertions(SVA), into temporal logic. Two prominent formalisms are commonly employed to describe temporal logic: *Computational Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). CTL comprises atomic formulas such as True/False, propositional operators (e.g., and, or), modal operators (E for "there exists a path" and A for "for all paths"), and temporal operators (X for "next," G for "globally," F for "finally," U for "until"). CTL's semantics are grounded in the *Kripke* model, which provides the framework for assessing system behavior in relation to its specified properties.

A Kripke model is a quintuple $K = (S, S_0, R, A, l)$ with: [93]

- S: Represents a finite set of states.
- S_0: Signifies a set of initial states, with $S_0 \subseteq S$.

- R: Defines the transition relation, $S_0 \subseteq S \times S$, which encompasses all potential state transitions within the model.
- A: Encompasses a set of atomic formulas, denoted as $A = \{p, q, \ldots\}$, each capable of assuming either a true or false value.
- l: $A \rightarrow 2^S$: A valuation function that specifies for every formula in A the set of all states in S for which the atomic formula is valid.

A Kripke model can be conveniently derived from the FSM model of a digital circuit. Nevertheless, there are some drawbacks of *Model Checking*. Two primary limitations include the issue of state explosion, where the state-space grows exponentially, and its limited applicability to circuits with substantial sequential depth. However, *Symbolic Model Checking* can be effective into solving these limitations. Symbolic Model Checking employs a Boolean representation for the finite state machine. It achieves this by substituting explicit state representation with Boolean encoding. This strategic shift allows Symbolic Model Checking to effectively address the state explosion problem, making it capable of handling significantly larger designs compared to explicit state model checking. The process of transforming the state machine through Boolean encoding is commonly referred to as state-space traversal. Symbolic model checking often leverages proof methods such as Binary Decision Diagrams (BDD) and Satisfiability (SAT) to handle the symbolic representation of the model. These techniques enable more efficient verification and analysis of complex designs, offering a potential solution to the challenges posed by state explosion and large sequential depth in design verification [34, 36].

Bounded Model Checking

BMC combines traditional Model Checking principles with Satisfiability (SAT) solvers to enhance efficiency in the verification process. This technique involves the unrolling of the FSM representation of the design in a specified number of times, denoted as k. Unrolling essentially includes replicating identical circuit representations k times, with each roll and its subsequent roll having a distance of 1 unit in terms of time. During this process, the only variables left unconstrained are the inputs, outputs, and the initial state "s0," except when explicit constraints are applied. For any value of k greater than 0, the current state input of the circuit copy is connected to the previous copy's state output. Property verification is translated into a SAT problem, and the property is considered satisfied if the negation of the property fails to hold. BMC initiates its verification from timepoint $t = 0$ and aims to prove the property for k clock cycles, where $0 \leq t \leq k$. To ensure the proof holds for all reachable states, the choice of k must be sufficiently large, typically equal to or greater than the sequential depth of the automaton. The sequential depth denotes the minimum number of cycles required to reach every state within the machine at least once. While it may not always be practical to select a very large k, BMC offers distinct advantages, especially in addressing complex verification challenges within the industry as BMC tools tend to identify counterexamples more quickly and with reduced memory usage [93].

Interval Property Checking (IPC) represents a specific subset of formal verification techniques, sharing some similarities with BMC. However, an essential distinction between the two methods lies in their initiation points for property verification. In BMC, the verification process commences at the initial timepoint $t = 0$, meaning it starts from a well-defined initial state. On the other hand, IPC initiates property checks at a symbolic time t rather than from the fixed

$t = 0$. This symbolic timepoint signifies that properties do not need to start from an explicit initial state. Instead, they can initiate from any potential state that aligns with the assumptions embedded within the property. IPC employs a similar logical framework as BMC, seeking to verify the property over a specified duration of k clock cycles. The primary distinction is the flexibility of the starting point: $t \leq t_i \leq t + k$, where t_i represents a point within the symbolic time interval. Similar to BMC, the verification process in IPC relies on the absence of a satisfiable solution for the negated property to assert its validity. This means that if there is no feasible solution that satisfies the negation of the property, then the property is considered valid [93].

Formal Equivalence Checking

Formal Equivalence Checking (FEC) is a crucial technique employed in the implementation phase of a design to prove the absolute equivalence of two distinct designs using mathematical reasoning. Unlike many other verification methods, FEC substantially minimizes or even eliminates the need for manual intervention by Verification Engineers. FEC relies on the formal verification tool's capacity to process information related to the two designs under verification. FEC can be used to compare designs into different abstraction levels, thus these designs may be provided to the tool in various forms, including RTL descriptions or gate-level netlist representations of the circuit.

In the scope of FEC, two prominent types are extensively employed within the industry [119]: Combinatorial Equivalence Checking and Sequential Equivalence Checking. Combinatorial Equivalence Checking serves as a valuable tool for comparing two distinct versions of the same design, each existing at different levels of abstraction. In contrast, Sequential Equivalence Checking is employed to ascertain whether two distinct models produce identical output sets for an equivalent set of inputs across all time points. Unlike Combinatorial Equivalence Checking, Sequential Equivalence Checking does not necessitate the internal nodes of the designs to be structurally identical.

Relevance to this thesis

This thesis relies on the application of formal methods. FEC plays a pivotal role in verifying the accuracy of the model transformation process. Concurrently, Property Checking serves as a crucial tool for verifying the behavior of fault injectors introduced into the design through the model transformation procedure. Additionally, Property Checking finds extensive application in the domain of safety verification for processor cores.

Chapter 4

A Generic Approach for Fault Handling

Fault injection is necessary to evaluate the system's robustness and asses its fault tolerance and resilience. Through deliberate fault introduction, designers gain valuable insights into the system's behavior, uncovering hidden weaknesses and vulnerabilities in both the system's design and implementation. Typically, the system undergoes various fault injection campaigns according to the final robustness requirements. Due to the random nature of faults, commonly, fault injection campaigns are implemented with a statistical and probabilistic approach. Additionally, in-depth analysis of specific faults requires the application of direct and systematic fault injection. These diverse fault injection requirements necessitate an automated and efficient fault injection tool capable of bridging the gap between the efforts invested and the final fault injection campaign. Typically, commercial and open-source fault injection tools are designed to accept inputs, such as the fault list, signal strobes list, and test cases. The manual writing of inputs for the fault injection process can be cumbersome and time-consuming, making automation of this task highly recommended. Automating the process would streamline fault injection and improve efficiency.

The work presented in thesis provides an automated and versatile framework designed to generate diverse fault injection campaigns and at the same time to close the gap between the fault specifications and the fault injection process with minimal efforts. The framework provides a vendor-independent solution, thus all Verilog/SystemVerilog-based simulators/emulators can be utilized. A general overview of the fault handling framework is depicted in Figure 4.1. The framework backend relies on the model-driven code generator framework, *Metagen* (see Chapter 2 for more details). The fault handling framework is composed by two sub-flows such as: (i) *Fault Injection Handler*, and (ii) *Fault Injection Documentation*.

The core element of the Fault Handler is the *MetaFI* metamodel (FI stands for Fault Injection) that specifies the type of the fault injection campaign as well as the particular outputs/registers that should be analyzed during the campaign. At first, the process involves a *reader* that reads the design model (MoD) and translates the fault list into a formal model. Once the fault list has been formalized, the next step involves defining the fault injection features via an intermediate model, FI-model. This model contains all the required information to perform the fault injection campaign such as fault attributes and design strobes (signals) to analyze. Finally, a template engine translates the model into the final Fault Handler testbench in two different views : SystemVerilog and Verilator-based C++.

The FI-model serves as an input for the documentation generation flow MetaDOCU. Meta-

DOCU is the core metamodel of the documentation generation flow and in a similar fashion like all Metagen-based flows, MetaDOCU translates the fault injection model into structured documentation views such as Portable Document Format (PDF).

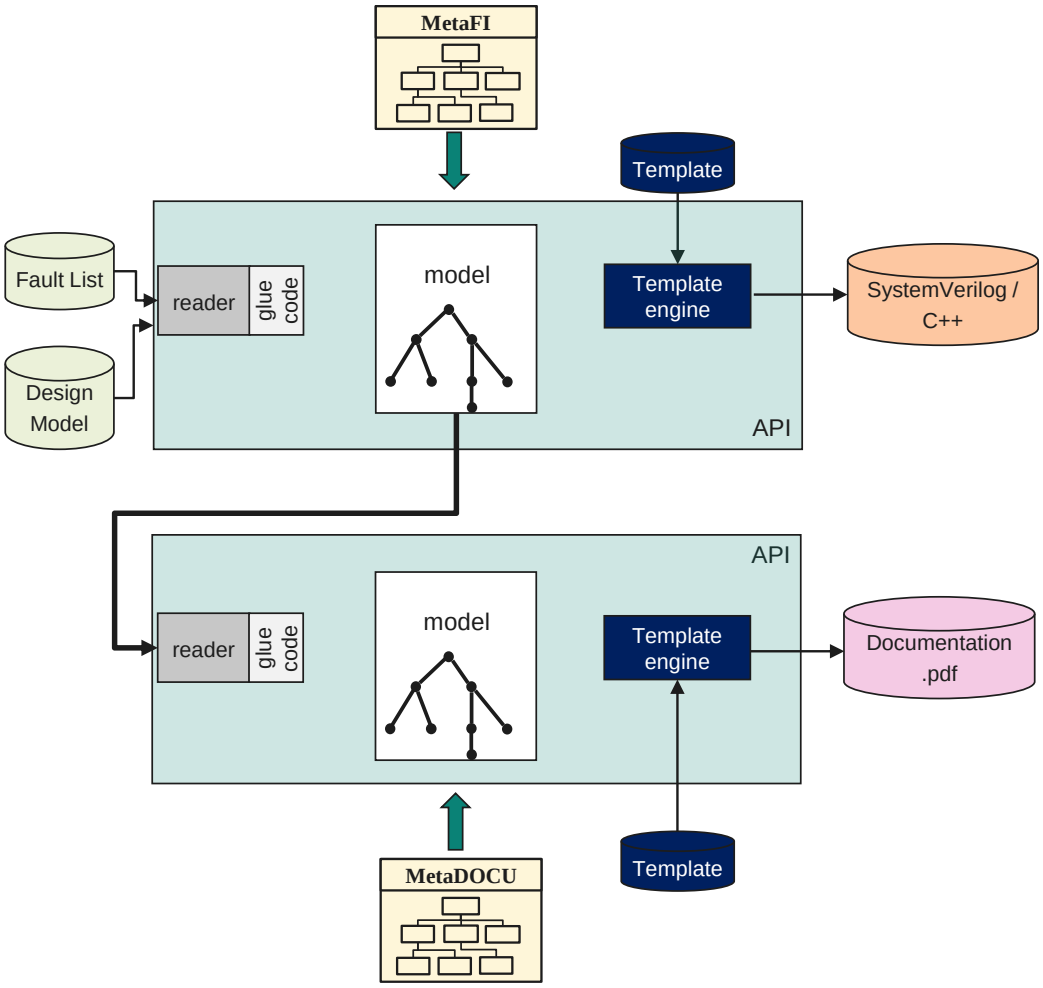


Figure 4.1: Fault handling framework

In the subsequent sections, a comprehensive description of the Fault Handler flow, alongside the documentation flow, is provided.

4.1 The Generic Fault Handling

The fault handler is the core element of the automated fault injection flow providing all necessary means to handle all fault injection attributes and features. Similar to all Metagen-based frameworks, the fault injection framework generates fault simulation/emulation testbenches in a structured, parametrized and model-driven fashion. The framework is structured in three layers (similar to MDA) such as specifications layer, model layer and view layer. This section provides initially an overview of the fault handler framework and later explains in details its features. At

the same time, it illustrates respective algorithms that are utilized to create the model. In the end, the section provides examples of the generated testbenches according to the model.

4.1.1 Specifications layer

The integral part of the fault handler framework is the MetaFI metamodel, shown in Figure 4.2. The metamodel describes all the necessary features that a fault injection campaign requires such as : (i) *Fault Controller*, (ii) *Fault List*, and (iii) *Fault Analyzer*. The rootnode of the metamodel has relations to different sets of classes that are categorized according to the aforementioned features.

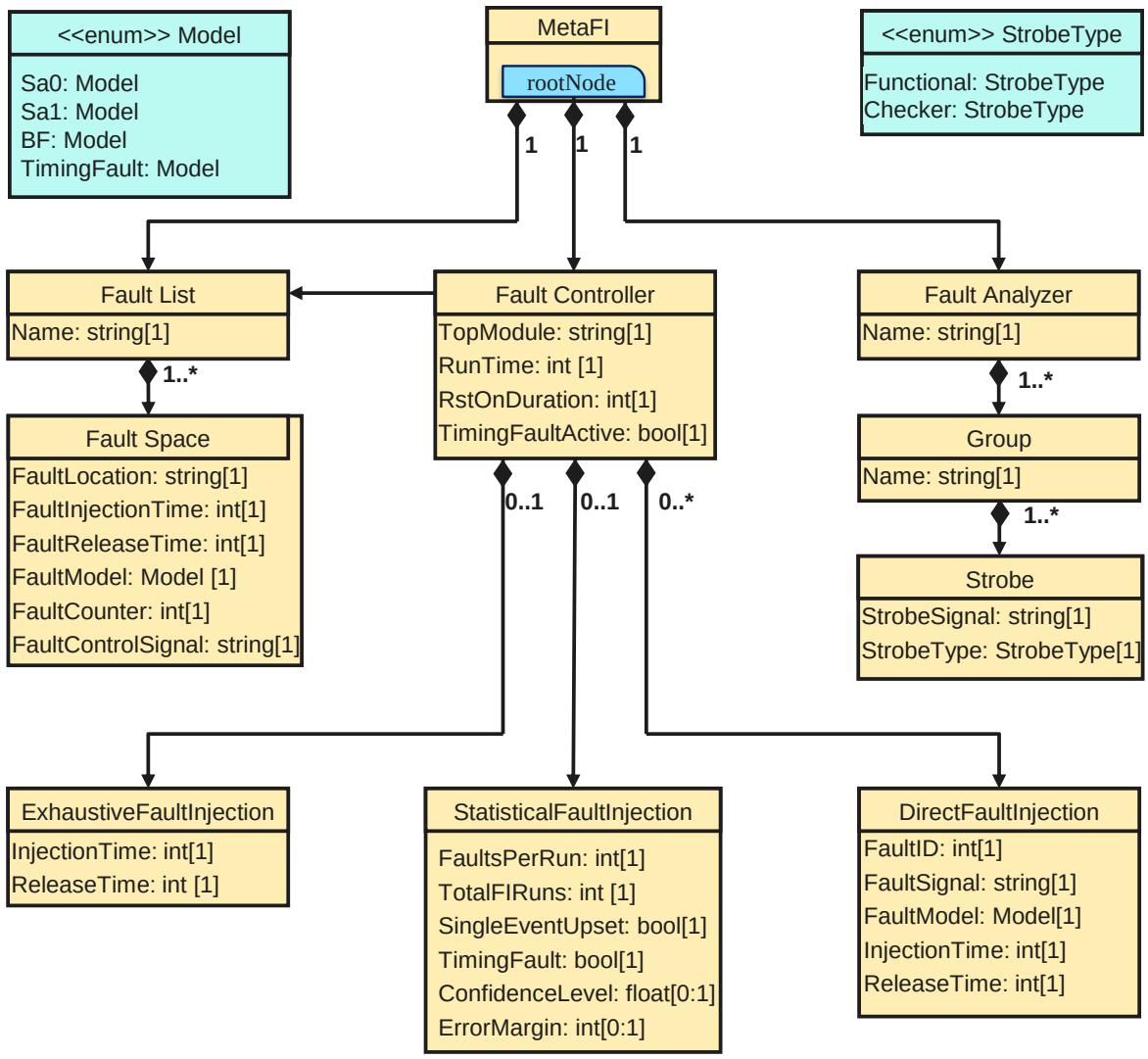


Figure 4.2: Fault handling metamodel

Fault Controller

The *Fault Controller* encompasses a set of classes and class attributes designed to specify the purpose of fault injection and fault models. Within the Fault Controller class, there exist three key attributes: *TopModule*, which denotes the top module of the RTL for conducting fault injection; *RunTime*, determining the number of clock cycles for the simulation/emulation run; *RstOnDuration* specifying the duration of active reset; and *TimingFaultActive* indicating whether timing faults are supported. Additionally, the Simulation Controller class has a composition relation to three other classes: *StatisticalFaultInjection*, *DirectFaultInjection*, and *ExhaustiveFaultInjection*.The aforementioned classes constitute a comprehensive representation of the fault campaign types commonly encountered in fault injection as following:

- **Statistical Fault Injection (SFI)**: a design with N nodes consists of 2N possible single stuck-at fault models. For modern designs, the number of single stuck-at is too high. To improve the run time, fault injection employs various techniques like SFI, aiming to mitigate the computational effort involved in fault injection. In this technique, a random subset of faults is selected from the entire set of faults. Subsequently, the faults within this random sample are subjected to simulation/emulation. The sample coverage, i.e., the ratio of detected faults to all faults in the sample, is used as an estimate of the fault coverage in the complete fault set. As a last step, the evaluation of the margin of error and confidence interval is calculated. *StatisticalFaultInjection* class of the metamodel contains the following attributes: *FaultsPerRun*, indicating the number of faults to inject in a single run, i.e., fault multiplicity; *TotalFIRuns*, denoting the number of independent fault injection runs; *SingleEventUpset* determining whether only bit flips at memory cells should be injected; *TimingFault* determining whether only timing (delay) faults should be injected; *ConfidenceLevel* and *ErrorMargin*, representing sampling features that can be calculated according to *TotalFIRuns*. A more detailed explanation will follow later.

- **Direct Fault Injection (DFI)**: according to ISO26262, DFI is a technique specifically designed to assess the failure mode resulting from random errors in the functional logic of the design and to verify the expected behavior of the intended safety mechanism. It is important to note that DFI is not employed to evaluate the coverage or effectiveness of the safety mechanism in terms of fault detection or mitigation. Instead, its primary focus lies in validating the system's response to effects of the injected fault at specific locations, specific injection time and specific fault model. The class *DirectFaultInjection* encompasses the following attributes: *FaultID*, listing the set of the faults to inject per run, i.e., faults that have the same ID will be injected at the same simulation/emulation run; *FaultSignal*, determining the fault location; *FaultModel* indicating the selected fault model according to the enumeration class *Model*; *Model* class contains four different fault models that can be injected such as (i) stuck-at-0 (sa0), (ii) stuck-at-1 (sa1), (iii) bit-flip (BF) and timing fault; *InjectionTime* and *ReleaseTime*, specifying the injection and release time of the fault respectively.

- **Exhaustive Fault Injection (EFI)**: an EFI campaign, also known as systematic fault injection, involves sequentially injecting stuck-at-faults at every signal within a specific region of interest, e.g., commonly in DFT techniques. By doing so, a set of critical bits can be identified. These are the bits that must retain their correct state for the design to

behave as intended. Systematic fault injection is instrumental in revealing vulnerabilities and sensitivities in the design's behavior, thereby helping to enhance its fault tolerance and overall reliability [62]. *ExhaustiveFaultInjection* class contains only two attributes that are required to specify correct timing of the faults such as: *InjectionTime* and *ReleaseTime*.

Fault List

The *Fault List* class includes the description of injected faults, i.e., the fault space. The class exhibits a one-to-many relationship (1..*) with the *Fault Space* class, implying that each fault within the *Fault Space* is considered for fault injection. As explained in Chapter 3, a fault space describes all the attributes and features of the injected faults. The attributes of *Fault Space* are listed as follows: *FaultLocation*, *FaultInjectionTime*, *FaultReleaseTime*, *FaultModel*, *FaultCounter*, i.e., counting the injection run, and *FaultControlSignal* determining the signal that control the specific fault location.

Fault Analyzer

The *Fault Analyzer* constitutes the section of the metamodel responsible for gathering and examining data obtained from fault injection runs. This class has a one-to-many (1..*) relationship with the *Group* class. The class *Group* represents modules and submodules within the design, whose outputs are intended for analysis. It establishes a one-to-many relationship (1..*) with the *Strobe* class, which defines all output signals to be analyzed. The *StrobeSignal* attribute identifies the specific output signal. Additionally, the *StrobeType* attribute distinguishes the type of output, categorizing it as either a functional strobe (representing the functional output of the design) or a checker strobe (reflecting the output of a safety mechanism). The enumeration class *StrobeType* indicates the types of the outputs.

4.1.2 Model layer

The metamodel, shown in Figure. 4.2 (see page 51) describes the attributes and features of the common fault injection campaigns through metamodel objects, their attributes and the relation between objects. The Model-of-Things (MoT) is an instance of the metamodel and is created using a GUI that is provided by the underlying Metagen framework. Generally, the engineer populates the MoT data manually. During a fault injection campaign, the engineer would need to to manually fill all the data regarding the *Fault Controller* and all its related classes, i.e., *ExhaustiveFaultInjection*, *StatisticalFaultInjection*, *DirectFaultInjection*. It is not feasible to populate *Fault List* and *Fault Analyzer* data due to the size of complex designs, i.e., thousands or millions fault injections and thousands of signals that can be analyzed. For this reason, a model-to-model DSL-based transformation script transforms the MoT into a less abstract intermediate model known as Model-of-Fault-Injection (MoFI). The MoFI is an instance of the MetaFI metamodel and contains all the necessary information to perform the fault injection. *Fault List* and *Fault Analyzer* are automatically populated through the transformation script known as Template-of-Fault-Injection (ToFI). ToFI utilizes built-in Metagen APIs such as setters and getters to read and set particular values to the model. The ToFI is structured into two main blocks: (i) Fault Analyzer block, and (ii) Fault List block.

In the *Fault Analyzer block* of ToFI, certain algorithms are utilized to read the original design MoD, i.e., the model of the design where the fault injection will be performed. This is necessary to get design internal signals information to set up the strobes of MoFI. Algorithm 1 displays the procedure to read the MoD and set functional strobes and checker strobes of the MoFI. The algorithm takes as inputs the MoD, FI MoT and information regarding implemented safety mechanisms in the design, e.g., their names. The first step of the algorithm is to check which components of the MoD should be selected to be analyzed as specified in the *Fault Analyzer*, lines 2-4, by iterating thorough a *for* loop through all groups and MoD components. Each component of the MoD is compared to the *Groups* attribute of the MoT and all the output ports of the components are set as strobe signals, lines 5-6. Then a check is performed to classify the output ports as functional or checker strobes, lines 7-10 and setting correct signals to the model.

Algorithm 1 Algorithm to set functional and checker strobes

Input: MoD, FI_MoT, safety_mechanisms
Output: functional strobes and checker strobes

```
1:  function SET_STROBES()
2:      for group in FI_MoT.Fault_Analyzer.Groups() do
3:          for component in MoD.Components() do
4:              if group = component then
5:                  for port in component.getOutPorts() do
6:                      FI_MoT.Fault_Analyzer.Groups.StrobeSignal.set(port)
7:                      if port in safety_mechanisms then
8:                          FI_MoT.Fault_Analyzer.Groups.StrobeType.set(Checker)
9:                      else
10:                         FI_MoT.Fault_Analyzer.Groups.StrobeType.set(Functional)
11:                     end if
12:                 end for
13:             end if
14:         end for
15:     end for
16: end function
```

The *Fault List block* utilizes three different algorithms to automatically populate the Fault List according to the type of fault injection campaign, respectively EFI, SFI and DFI. This block takes the fault list description as an input and sets the formalized fault list and fault space accordingly.

During an EFI campaign type, both stuck-at fault models must be injected at all signals in the fault list. Algorithm 2 illustrates the procedure to set the fault list. An internal integer *cnt* (line 2) serves as a counter for injection run and two variables *Fi_inj_time*, and *Fi_rel_time* read the fault injection and release time from the attributes of *ExhaustiveFaultInjection* class. A *for* loop iterates through all signals of the fault list (line 5), and another *for* loop considers fault models to inject (line 6). As next step, a fault space is added for every signal and fault model (line 7), i.e., if a design has 10 signals (fault locations), then 20 independent fault spaces will be created. Lines 8-14 simply set the respective fault space values according to the input data.

For a DFI campaign type, the fault list should include only certain fault injection features

Algorithm 2 Algorithm to set fault list for an EFI campaign

Input: fault_list, FI_MoT
Output: Fault_List

```
1: function SET_EFI_FAULT_LIST()
2:     int cnt = 1
3:     FI_inj_time = FI_MoT.Fault_Controller.ExhaustiveFaultInjection.InjectionTime()
4:     FI_rel_time = FI_MoT.Fault_Controller.ExhaustiveFaultInjection.ReleaseTime()
5:     for signal in fault_list do
6:         for model in [sa0, sa1] do
7:             FI_MoT.Fault_List.addFault_Space()
8:             FI_MoT.Fault_List.Fault_Space().setFaultLocation(signal)
9:             FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(FI_inj_time)
10:            FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(FI_rel_time)
11:            FI_MoT.Fault_List.Fault_Space().setFaultModel(model)
12:            FI_MoT.Fault_List.Fault_Space().setFaultCounter(cnt)
13:            FI_MoT.Fault_List.Fault_Space().setFaultControlsignal(signal.control())
14:            cnt = cnt+1
15:        end for
16:    end for
17: end function
```

Algorithm 3 Algorithm to set fault list for an DFI campaign

Input: fault_list, FI_MoT
Output: Fault_List

```
1: function SET_DFI_FAULT_LIST()
2:     int cnt = 1
3:     for dfi in FI_MoT.Fault_Controller.DirectFaultInjection() do
4:         FI_ID = dfi.FaultID()
5:         FI_location = dfi.FaultSignal()
6:         FI_model = dfi.FaultModel()
7:         FI_inj_time = dfi.InjectionTime()
8:         FI_rel_time = dfi.ReleaseTime()
9:         FI_MoT.Fault_List.addFault_Space()
10:        FI_MoT.Fault_List.Fault_Space().setFaultLocation(FI_location)
11:        FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(FI_inj_time)
12:        FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(FI_rel_time)
13:        FI_MoT.Fault_List.Fault_Space().setFaultModel(FI_model)
14:        FI_MoT.Fault_List.Fault_Space().setFaultCounter(cnt)
15:        FI_MoT.Fault_List.Fault_Space().setFaultControlsignal(FI_location.control())
16:        if FI_ID ≠ previous(FI_ID) then
17:            cnt = cnt+1
18:        end if
19:    end for
20: end function
```

according to the *DirectFaultInjection* attributes. During this campaign, only a desired subset of all faults will be injected. Algorithm 3 illustrates the procedure to set the fault list. Similarly to the EFI campaign, an integer variable counts the injection run and then various variables store *DirectFaultInjection* attributes, lines 4-8, for all DFI instances. These attributes are used to set *Fault List* attributes as shown in lines 9-15. The counter is incremented only if the fault ID is different from the previous one,lines 16-17, because similar ID faults would be injected at the same run.

The SFI campaign is the most widely used technique complying with ISO26262 standards. For a better understanding let us revisit the basic mathematical definitions of a SFI campaign. Fault sampling is generally used and the accuracy of the estimated fault coverage relies on the sample size, which represents the absolute number of faults in the sample. A larger sample size leads to a reduced error bound in the coverage estimate. The determination of the sample size is based on the desired accuracy level for estimating the coverage. Leveugle et al. [98] propose methods for estimating and quantifying the error associated with statistical fault injection, providing valuable insights for confidence in the fault injection result. The main hypothesis is that the underlying computations presume that the properties of the entire fault injection population, i.e., specifically all potential fault models occurring at any clock cycle, conform to a normal distribution. The process of random fault sampling ensures that each individual fault configuration at a specific cycle within the initial population has an equal probability of being selected for the sample. This is achieved by using a uniform distribution for the random selection [98]. Thus the number of n faults to inject per campaign can be calculated by [98]:

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{Z^2 \times p \times (1-p)}} \tag{4.1}$$

where:

- N is initial population size, i.e., all possible faults to inject at every clock cycle,
- p is the standard error, i.e., standard deviation of its sampling distribution,
- e is the margin of error, and
- Z is confidence level parameter, i.e., the probability that the exact value is actually within the error interval.

Typically N is very large, thus the equation 4.1 can be simplified by assuming $N- > \infty$ as follows:

$$n = \lim_{N \to \infty} f(N) = \frac{Z^2}{e^2} \times p \times (1-p) \tag{4.2}$$

As an example, let us consider a design that has 10^4 fault locations and simulation will run for 10^6 clock cycles. N is calculated as 10^{10} and can be assumed as very large, i.e., ∞. The number of faults to inject with given error margin and confidence level is shown in Table. 4.1. As can be seen from the table, even for very large fault injection population, only 23784 fault injections are required to achieve 99.8% confidence of fault injection results with 1 % error margin.

Table 4.1: Total number of fault injections according to error margin and confidence level

	95% confidence z = 1.96	99%confidence z = 2.5758	99.8% confidence z = 3.0902
e = 5%	n = 384	n = 663	n = 955
e = 1%	n = 9581	n = 16519	n = 23874

Algorithm 4 Algorithm to set fault list for an SFI campaign

Input: fault_list, FI_MoT
Output: Fault_List

```
1:  function SET_SFI_FAULT_LIST()
2:      int cnt = 1
3:      int cnt_per_run = 1
4:      sfi = FI_MoT.Fault_Controller.StatisticalFaultInjection
5:      FI_faults_per_run = sfi.FaultsPerRun()
6:      if sfi.hasTotalFIRuns() then
7:          FI_sample_size = sfi.TotalFIRuns()
8:      end if
9:      FI_SEU = sfi.SingleEventUpset()
10:     FI_Timing = sfi.TimingFault()
11:     if sfi.hasConfidenceLevel() then
12:         FI_confidence_level = sfi.ConfidenceLevel()
13:     end if
14:     if sfi.hasErrorMargin() then
15:         FI_error_margin = sfi.ErrorMargin()
16:     end if
17:     sample = SAMPLE_SIZE(FI_sample_size, FI_confidence_level, FI_error_margin )
18:     print (FI_sample_size, FI_confidence_level, FI_error_margin )
19:     while cnt ≤ sample do
20:         while cnt_per_run ≤ FI_faults_per_run do
21:             FI_MoT.Fault_List.addFault_Space()
22:             FI_MoT.Fault_List.Fault_Space().setFaultLocation(random(FL_signals))
23:             fault_loc = FI_MoT.Fault_List.Fault_Space().FaultLocation
24:             FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(random(sim_time))
25:             FI_MoT.Fault_List.Fault_Space().setFaultInjectionTime(random(sim_time))
26:             if FI_SEU = True then
27:                 FI_MoT.Fault_List.Fault_Space().setFaultModel(BF)
28:             else if FI_Timing = True then
29:                 FI_MoT.Fault_List.Fault_Space().setFaultModel(TimingFault)
30:             else
31:                 FI_MoT.Fault_List.Fault_Space().setFaultModel(random(model))
32:             end if
33:             FI_MoT.Fault_List.Fault_Space().setFaultCounter(cnt)
34:             FI_MoT.Fault_List.Fault_Space().setFaultControlsignal(fault_loc.control())
35:             cnt = cnt+1
36:         end while
37:     end while
38: end function
```

Algorithm 4 illustrates the procedure to set the fault list for an SFI campaign. The automated SFI campaigns follows the fault sampling principles by setting the necessary attributes in the *StatisticalFaultInjection* class. The user has the flexibility to set only two of three fault sampling

attributes, i.e., two attributes from confidence level, error margin, and the sample size (number of faults to inject). The automated flow would calculate the remaining attribute using built-in Python math libraries. As can be seen in the algorithm, lines 2-3 are setting some internal counter variables, respectively to count total runs (line 2) and total injections per run (line 3). Lines 4-16 read and stores class attributes into different variables, while line 17 calculates the number of total injections according to equation 4.2 and line 18 prints the metrics. Line 19 is iterating through the total sample that was previously calculated and line 20 is iterating through the number of faults per run. Lines 21-25 are simply setting the Fault List values using built-in random functions. Lines 26-32 set the particular fault model; if FL_SEU or FL_Timing variables are set to true only these models will be injected, otherwise a random fault model is injected. Lines 33-35 are similar to previous algorithms.

Using all the aforementioned algorithms, the transformation script ToFI creates the model of fault injection, MoFI, that contains all the information to generate different views.

4.1.3 View layer

The view layer represents the final layer of the automated fault handling framework. In this layer, the previously defined MoFI elements are mapped to a specific HDL-based language to create a testbench to perform fault injection. As the MoFI is designed to be platform language agnostic, they can be mapped to various languages. The current supported languages in the generation flow are Verilog/SystemVerilog and C++.

After creating the model through the algorithms presented in the previous section, the view layer involves automating the generation of the testbench. To accomplish this, a Mako template was developed to convert the fault injection model's information into the target code. An overview of the testbench structure is depicted in Figure 4.3.

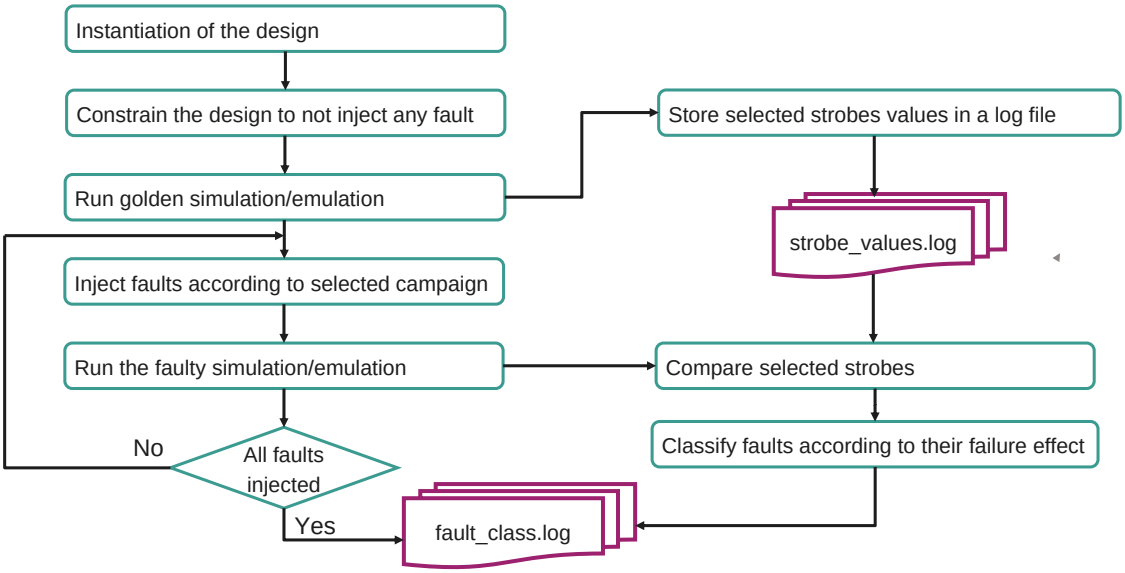


Figure 4.3: Fault handling testbench structure

The testbench starts by instantiating the design and by configuring the environment to its initial state. It sets the fault control signals to their default values to not inject any fault and

initializes all relevant variables that are required to count the injection runs. Next, a sequential clocked process starts that simply simulates/emulates the golden design, i.e., the design without any injected fault, for as many clock cycles as determined in the MoFI. Then, a log-file is generated that stores the values of the selected strobes at every clock cycle. After this process is finished, a *for* loop is created that injects faults sequentially, one after the other. This loop iterates as many times as specified in the MoFi, i.e., according to the fault injection campaign algorithms. During each loop step, the values of the selected strobes of the MoFI are compared to the stored values in the log-file. The comparison classifies the faults according to their effects: (i) safe fault, i.e., the fault does not propagate and it is not detected/corrected by the safety mechanism, (ii) safe detected fault, i.e., the fault does not propagate but it is detected/corrected by the safety mechanism, (iii) failure detected, i.e., the fault propagated but detected by the safety mechanism, and (iv) failure undetected, i.e., the fault propagated but is not detected by the safety mechanism. After evaluating all the injected faults per run, the environment is reset to its initial state to prepare for the next fault evaluation. Once all the injected faults have been processed, the final results of the fault injection campaign are recorded in the log-file. These results include the total number of faults, the count of safe faults, safe undetected faults, failure detected faults, and failure undetected faults. Subsequently, the golden and log-files are closed, and the fault injection campaign concludes. Algorithm 5 displays the pseudocode of the generated testbench.

Algorithm 5 Generated testbench pseudocode

```
1:  instantiate()
2:  initialize_variables()
3:  no_fault()
4:  generate_golden_log_files()
5:  //golden run
6:  for clock_edge do
7:      while time_cnt < sim_time do
8:          simulate()
9:          store_strobes()
10:         time_cnt = time_cnt + 1
11:     end while
12: end for
13: //fault injection and fault classification
14: while sim_cnt < total_sim do
15:     for clock_edge do
16:         while time_cnt < sim_time do
17:             inject_faults()
18:             compare_classify_strobes()
19:         end while
20:         time_cnt = time_cnt+1
21:     end for
22:     reset_variables()
23:     sim_cnt = sim_cnt + 1
24: end while
```

As can be seen from the algorithm, lines 1-2 are required to instantiate the design and initialize the variables. Line 3 is constraining the design to not inject any fault and line 4 is a procedure that generates the log-fileiles to store strobes information. The first process includes the golden run where the strobes value are stored at every clock cycle in the log-file, (lines 6-12). This process runs as long as defined in the *sim_time* variable that corresponds to the data from the MoFI. After the golden run is finished, the process of fault injection starts. A loop is executing (line 14) to make the process iterating as many times as specified in the MoFI, i.e., the total fault injections as defined from the fault injection campaign algorithms. The strobes are compared to the golden strobe values and the faults are classified according to their effect (lines 15-21). After a fault injection run is finished, the variables are reset (line 22), and the simulation count is increased (line 23). The Mako template generates the pseudocode in Verilog/SystemVerilog and C++ language.

The model-driven fault handling framework provides a fully automated and structured solution to perform various fault injection campaigns. Furthermore, it formalizes informal fault injection specifications and at the same time provides a generic simulator/emulator independent solution.

4.2 Generic Documentation of Fault Injection Campaigns

Documentation plays a crucial role in fault injection, serving as an essential aspect of the process. Proper documentation involves recording all relevant details related to the fault injection campaign, such as the types of faults injected, their locations, and the fault analysis outcomes. In this thesis, a novel model-driven documentation generation framework has been developed to address the need for comprehensive and structured documentation in fault injection campaigns. By utilizing the underlying models, the framework efficiently captures and organizes critical information related to fault injection experiments, including fault location, fault models, campaign type, and other relevant data.

The documentation generation framework is generic, making it applicable for diverse document generation needs. The framework's versatility allows it to be effectively utilized for generating documentation across various domains and contexts, beyond the scope of fault injection alone. Nevertheless, in the following, the focus will be specifically on fault injection documentation.

4.2.1 Overview of the Documentation Generation Framework

Similar to all model-driven frameworks, the core element of the documentation generation is the metamodel depicted in Figure 4.4. The root node *MetaDOCU* has a relationship with the *Document* class which itself contains only the *Author* attribute. The *Document* class has relationship with two other classes such as *Format* and *DocumentItem*. One of the initial steps involves formatting the document, including aspects such as font selection and size adjustments through *Format* class. *DocumentItem* simply specifies all possible items that the generated document can have, e.g., title that is specified through *Title* class. There are several document item options (*DocumentItemOpt*) that can be specified through various classes such as *List*, *Figure* and *Table*. Each of these classes contains different attributes to configure all document items.

Through *TextBlock* class, the user can specify a new section of the document. Finally, the *TextItem* class simply contains the text that should be inside the TextBlock. Generally, the user can specify different formats for each section or text item. Finally, The *References* class serves the purpose of creating references to external sources within the document.

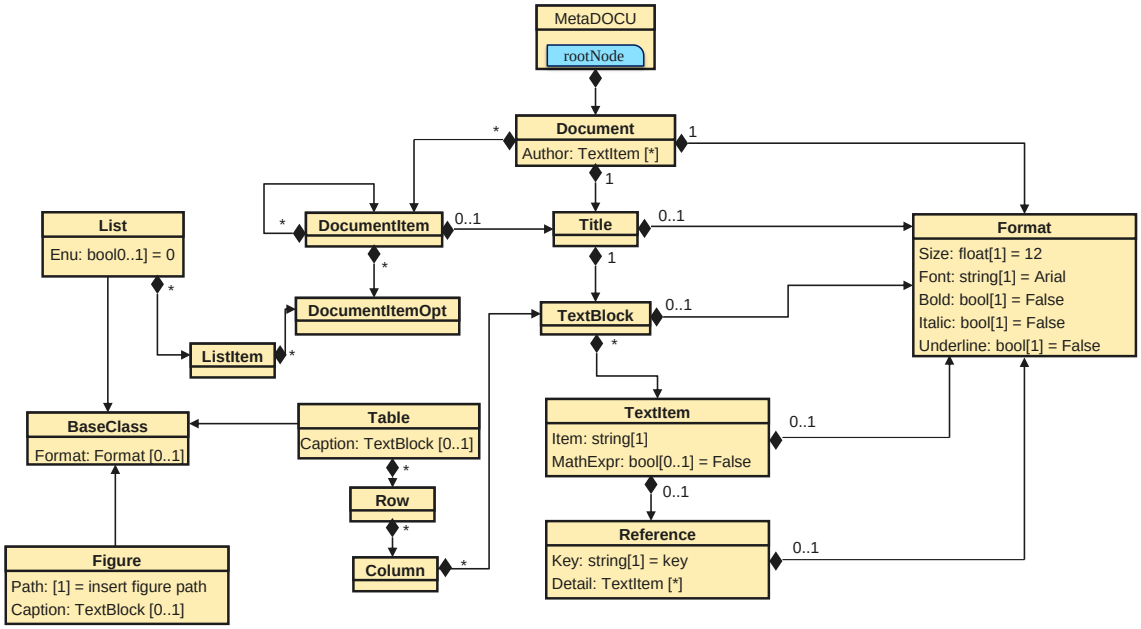


Figure 4.4: MetaDocu metamodel

After the documents specifications are captured via the metamodel, internal readers and template generators have been developed that generate the documents into different formats such as LaTex or Markdown.

4.2.2 Fault Injection Documentation Generation

The fault injection documentation is generated using a documentation generator that takes the MoFI as input. The MoFI is then transformed into a structured documentation model using a template designed specifically for this purpose. The simplicity of the documentation generator facilitates the development of the template. The template contains various functions to capture different fault injection features. Initially, a function is used to set the documentation title, document's format and the initial document items. Then, a section is created containing the general fault injection attributes such as information about the design, fault injection campaign type. Lastly, a table is created that contains all the information about the injected faults.

For illustration purposes, a snippet of an example of generated fault injection documentation is shown in Figure 4.5. The design where fault injection is performed is called *sample circuit* and then the simulation duration is noted. After that, the document shows the type of the fault injection campaign, and in the figure an EFI campaign is displayed. Next step consists of displaying the fault injection and release time. Then, the document displays the strobes that will be analyzed during simulation/emulation. In this case, the signals are *sample_circuit_out_1* and *sample_circuit_out_2*. After the general information is provided, a table is created that lists

every injected fault attributes. The first column displays the simulation count, and next all fault injection attributes are shown such as fault location, fault injection time, fault release time and fault model. Since an EFI campaign was selected, it can be seen from the figure that both stuck-at fault models are injected at two signals (*sig_1* and *sig_2*).

Fault injection is applied on: **sample_circuit**
Simulation duration is : **50 clock cycles**
The selected type of fault injection campaign: **EFI**
Faults are injected starting from timepoint: **20**
Faults are released at timepoint: **50**
The following signals will be analyzed:

- **sample_circuit.out_1**
- **sample_circuit.out_2**

Fault injection data:

Sim count	Fault location	Injection time	Release time	Fault model
1	sig_1	20	50	sa1
2	sig_1	20	50	sa0
3	sig_2	20	50	sa1
4	sig_2	20	50	sa0

Figure 4.5: Snippet of fault injection documentation

Chapter 5

Fault Simulation on Mixed Granularity RTL Models

The increasing functional complexity and transistor density of modern electronic systems have made them more vulnerable to both systematic and random hardware failures. Systematic failures occur during the product development cycle, while random failures happen during the system's field operation. In the automotive domain, where safety-critical electronic components are used, ISO 26262, the functional safety standard, mandates thorough analysis of all types of failures during the development process to ensure risks are below acceptable levels. This high level of complexity requires fast and trustworthy fault injection methods. *Fault simulation* is the widely employed fault injection technique that offers numerous advantages. One of its primary benefits is its cost-effectiveness compared to physical fault injection experiments or real-world testing. By conducting fault simulations in a virtual environment, engineers can detect and address potential faults early in the development process without the need for costly physical prototypes. This early fault analysis helps in avoiding expensive redesigns and improves overall development efficiency. Moreover, fault simulation contributes to an accelerated development process. Engineers can perform fault simulations iteratively throughout the design phase, allowing for faster identification of design flaws and quicker resolution of issues. This iterative approach enables a more agile development cycle, reducing the time-to-market and enhancing the overall product quality.

Fault simulation is a versatile technique that can be applied at various abstraction levels of a design, including transistor level, gate-level, and RTL. However, its most common and preferred application is at the gate-level representation of the design, as it provides accurate fault coverage [88, 125]. Fault simulation at gate-level abstraction allows for a detailed analysis of the behavior of individual gates and logic elements, providing precise fault modeling and accurate fault injection. Despite the benefits of gate-level fault simulation, such as detailed impact analysis of injected faults at the logic gates and registers, its simulation performance is slow. Therefore, fault simulation is also commonly applied at the RTL due to its higher performance compared to gate-level fault simulation and at the same time being less resource-intensive. Nevertheless, RTL fault simulation suffers from a less detailed analysis because the higher-level abstraction of RTL may not capture the same level of fine-grained details as gate-level simulation, which could result in overlooking certain fault behaviors. Binary simulation, i.e., 0/1 simulation, requires manipulation of certain bits to mimic the fault injection.

This thesis proposes a novel approach that performs fault simulation on mixed "coarse-grained" granularity RTL model. The approach overcomes the challenges of both gate-level and RTL fault simulation by combining the best of the two worlds. The fault simulation flow allows that only the design modules that are object of fault injection are represented at "fine-grained" gate-level granularity while the rest of the design is represented at original RTL granularity. The proposed fault simulation drastically improves gate-level fault simulation performance and at the same time is sufficiently accurate. The MetaRTL RTL generation flow streamlines the entire workflow, making it highly automated and efficient.

This chapter offers a comprehensive explanation of the fault simulation flow by presenting intricate details of all the techniques employed in the process. An overview of the proposed approach has been previously published at [79].

5.1 Overall Flow

The proposed approach involves employing a mixed granularity representation for the DUT to accelerate fault simulation. In this scheme, the modules targeted for fault injection are depicted with gate-level granularity, while the remaining sections of the design maintain an RTL representation, which is adequate for fault propagation analysis. Creating RTL design models manually, featuring distinct gate-level granularity modules, is both labor-intensive and susceptible to errors. To counter this, MetaRTL is integrated into the fault simulation approach. The work introduced in this thesis entails an expansion of the existing RTL generation flow to encompass the creation of RTL code at gate-level granularity.

The complete fault simulation flow is depicted in Figure 5.1. Initially, MetaRTL is utilized to generate the design. Subsequently, the functionality of the generated RTL design is verified through functional verification, following the methodologies outlined in Chapter 2, i.e., MetaProp. Functional verification is required to remove systematic faults, because they could hinder the effects of random faults. Following the functional verification, the generated RTL design undergoes synthesis to achieve its gate-level representation. The synthesis process is executed using a logic synthesizer, for example, Design Compiler [124] or Yosys [133]. The transformation of RTL to its gate-level equivalent is contingent on the specific technology employed, thus it adheres to a dedicated technology library. This library includes technology cells, each accompanied by important information encompassing aspects such as timing, power, area, and logic expression.

An implemented Python program, denoted as the *ToD Generator*, is utilized for extracting the gate-level netlist and library cell functionalities. This task is achieved through the utilization of HDL parsers, such as the Verific parser [131], which facilitate the extraction process. The unique functionalities associated with individual technology library cells are then mapped automatically to corresponding components within the MoD. This mapping is complemented by the utilization of built-in primitives, which serve to encapsulate the specific functionalities. The MoD is an instance of the MetaRTL metamodel, and as a result, a new MoD is generated that describes the design in the gate-level granularity (conforming to netlist) using MetaRTL primitives. For example, an AND gate of the netlist is represented using an RTL-based AND built-in primitive.

The next significant phase involves the process of *model transformation*, including the trans-

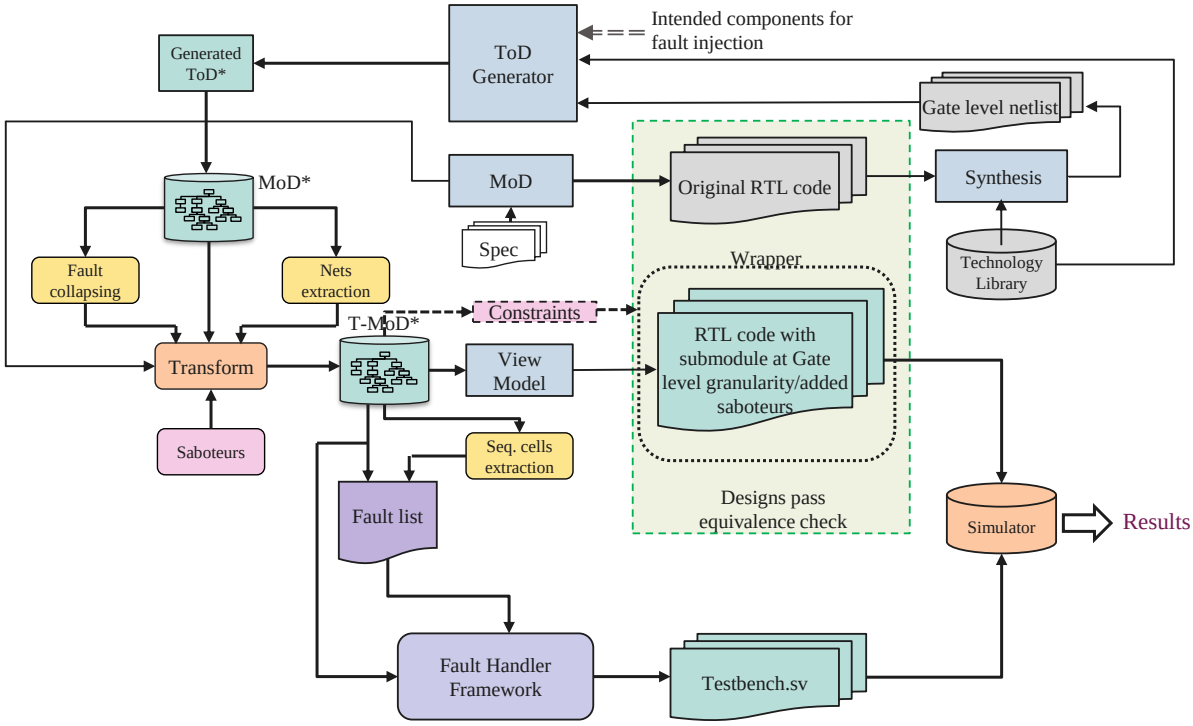


Figure 5.1: Overview of mixed-granularity fault simulation flow

formation of the generated MoD* to create a mixed-granularity model equipped with fault injection capabilities. Within this context, the component designated for fault injection within the original MoD is substituted with the analogous component from the generated MoD*. Subsequently, the model transformation operation is executed on the component model, involving the *insertion of saboteurs* across each component net. Moreover, to streamline the process, fault collapsing techniques can be employed to minimize the requisite number of fault injection locations.

This Transformed MoD* (T-MoD*), is then leveraged for the purpose of generating the design in a hybrid RTL model, combining different granularities. To elaborate, let us consider an instance where fault injection is to be executed solely on the Register-File of a processor. Synthesis is limited exclusively to the Register-File, resulting in the generation of a distinct MoD containing exclusive Register-File information. Subsequently, this distinct MoD is incorporated into the original MoD of the processor, thus building a mixed MoD characterized by: (i) a gate-level granularity for the Register-File, and (ii) traditional RTL representation for the remainder of the design. Afterwards, the model transformation procedure is implemented, and as a result a new processor is generated integrating fault injection capabilities. As a concluding step, an equivalence and property check is performed to compare the new generated design against the original version, effectively verifying the absence of bugs within the transformation process.

In parallel, the T-MoD and the fault list are used as inputs from the Fault Handling framework, as introduced in Chapter 4. This facilitates the generation of a customized testbench to execute various fault injection campaigns. The testbench and the mixed-granularity RTL models are utilized together to conduct fault simulation utilizing either commercial or open-source

RTL simulators. The resulting information from the simulations offers valuable insights into the effects of injected faults, thus contributing to a comprehensive understanding of the system's fault tolerance characteristics.

5.2 Background on Model Transformation

The MoD serves as the key model in the RTL generation process and captures the design microarchitecture, regardless of the platform or technology. It consists of *components*, *ports*, *connections* and *literals*. The components can be categorized as:

- Descriptive: design description style in the RTL generation flow,
- Behavioral: behavior description on a high level, e.g., Finite State Machines (FSMs),
- Sequential: sequential design elements, e.g., latches and flip-flops,
- Primitive: combinatorial basic logic gates, e.g., a logical OR gate.

The MoD can be considered as a tree-like data structure with the top module as its root and the components and sub-components forming the nodes and branches. Complex designs are typically built hierarchically, where high-level modules contain their respective sub-modules. For example, a pipelined processor core has the processor as the top-level module, the Execute stage as its sub-module, and the ALU as a sub-module of the Execute stage. In a similar way, the MoD describes the hardware hierarchically and can be considered as a hierarchical port graph [69, 56]. The formal representation of the design model enables its transformation by adding, replacing, or deleting elements, i.e., model transformation can be referred to as *graph rewriting*. Bavache et al. [30] presented the core operators to transform the MoD as following:

- **Locate**: locates all MoD elements that should be transformed.
- **Add**: adds new MoD elements to the previously located elements.
- **Delete**: deletes MoD elements and takes care of proper connections.
- **Replace**: replaces old MoD elements with new elements.

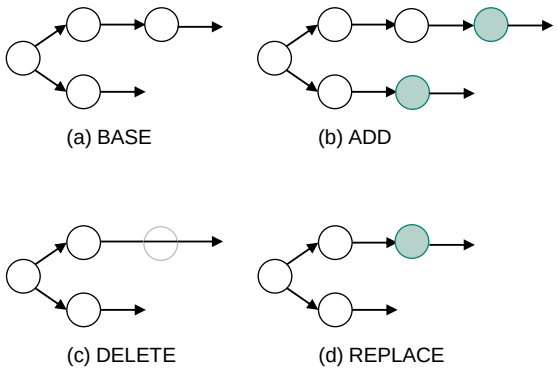


Figure 5.2: MoD transformation operators [30]

A visual representation of the MoD transformation operations is presented in Figure 5.2. The initial MoD that undergoes these transformations is depicted in Figure 5.2a. In Figure 5.2b, the base MoD is subjected to a transformation, introducing two new elements. Figure 5.2c displays the removal of an element from the base MoD, while Figure 5.2d portrays the substitution of an existing element with a new one.

Throughout this thesis, these transformation operators are extensively employed within the context of fine-grained MoD generation and the insertion of fault injectors.

5.3 Generation of Fine-grained Models

The principal objective underlying the generation of fine-grained models, specifically in the context of mixed granularity models, is the enhancement of fault simulation performance, and at the same time preserving a sufficient level of accurate fault coverage. This paradigmatic approach can be characterized as a form of *reverse engineering*, manifesting as a sequential transformation process: initial model generation, subsequent RTL synthesis, and eventual re-conversion into a refined model representation (model->RTL->model). In essence, this iterative process includes the creation of an alternative design model subsequent to the original RTL representation, effectively shifting the design pathway from model to RTL and then reverting to an advanced fine-granular model. The process consists of two main steps: (i) converting the generated RTL into a ToD, and (ii) creating a fine-grained MoD . In the following the process is explained in details.

5.3.1 Netlist-to-ToD

MetaRTL is utilized to generate the RTL from the initial specifications and then functional verification is performed to check against functional bugs, i.e., deviations from specifications. After this process is complete, synthesis of the RTL is performed, e.g., via Yosys synthesis tool, and the gate-level netlist of the design RTL is generated. In a gate-level netlist, the design is generally described using standard gate-level primitives, e.g., AND gate, OR gate, etc, and library cells that define the behavior and characteristics of the individual gates.

A *ToD generator* has been developed that converts the synthesis netlist into ToD constructs. The ToD generator is composed of two main parts: (i) library-ToD generator, (ii) netlist-ToD generator. These two individual generators are then combined to generate a final netlist ToD.

Library-ToD generator

The initial phase of generating a netlist-Tod entails the conversion of a technology library file into a *library-ToD* representation. A technology library can be conceptualized as a group of logic gates, each equipped with distinct attributes. These libraries, distributed by semiconductor foundries, encompass a comprehensive description of essential characteristics and operational details of each logic cell within a semiconductor vendor's library. A standardized technology library consists of combinational cells (e.g., logic gates), sequential cells (e.g., flip flops), and other cells (e.g., fillers). The *Verific library parser* is utilized to read and extract cell information from the library file such as: (i) cell name, (ii) combinatioral and latch cells information, (iii) pin information, and (iv) different attributes like pins direction, cells function etc..

A simplified version of a library cell is shown in Listing 5.1. The library contains information regarding the cell name (line 1), the cell's ports directions (lines 3-5), cell's funcionality (line 8) and the cell's pins (line 11-13). As can be seen from the functionality and the pins, *AN2X015* cell represents a two-input logical AND gate. The listing excludes other standard cell parameters like area, leakage power etc.

```
1   AN2X015: {
2     "dir": {
3       A: input,
4       B: input,
5       Z: output
6     },
7     "func": [
8       (A*B)
9     ],
10    "pin": [
11      A,
12      B,
13      Z
14    ]
15  }
```

Listing 5.1: Library cell example

A Mako template combined with a reader script, converts the above cell into a ToD structure where each cell of the library corresponds to an independent ToD python class. Listing 5.2 illustrates the ToD class for the example cell of Listing 5.1. In the library-ToD, the name of the class (line 1) corresponds to the name of the cell in the library file and then Python class constructs are created (lines 2-3). Then, all pins and their attributes e.g., direction and size, are defined in the lines 4-6. Each individual cell must be categorized as either a combinational logic cell or a sequential cell (such as a register or a latch). If it is a basic combinational logic cell, the extracted function is transformed into a dataflow expression, as illustrated in line 8.

```
1  class AN2X015(Dataflow):
2      def __init__(self, *args, **kwargs):
3          super(AN2X015, self).__init__(*args, **kwargs)
4          self.A = InPort(ObjProps=BitVec(1))
5          self.B = InPort(ObjProps=BitVec(1))
6          self.Z = OutPort()
7
8          self.Z = (self.A & self.B)
```

Listing 5.2: ToD of a library cell

Translation of sequential cells into the ToD consists of a few more steps. Predefined registers and latches within the MetaRTL framework are employed to translate register and latch cells. Each pin of the register and latch cell is linked to the corresponding ports of the predefined register and latch. The enable and reset pins are configured according to their retrieved pin features, determining their respective sensitivities. Additionally, some certain gate-level technology cells like registers and latches rely on an enable signal to trigger the fault, meaning the fault remains inactive if the cell is not enabled. If testing patterns disregard the enable signal of a memory cell, it could potentially lead to risky behavior within the design and impact dependability analysis. To address this scenario, in this thesis, library enabled sequential cells are transformed into simpler sequential cells. This involves connecting the input of these cells to the output of a multiplexer, where the enable signal serves as the selection signal. For instance, transforming an enabled D flip-flop is illustrated in Figure 5.3. Both circuits are equivalent – the cell is written only when the enable signal is activated. By introducing saboteurs at the input

of the cell, indicated in red in Figure 5.3, the cell's value can be flipped independently of the enable signal [75].

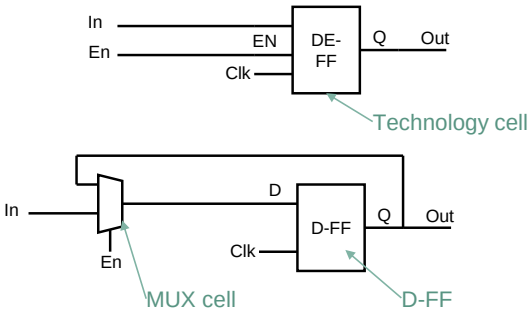


Figure 5.3: Conversion of enabled sequential cells to simple ones [75]

A snapshot of the ToD of a sequential test cell *FD1QSX010* is shown in Listing 5.3. Line 1-8 are very similar to the combinational cell definitions, i.e., cell name, class constructs and pins attributes. Line 10 represents how the test enable activates the register input and lines 11-14 simply represent a MetaRTL *Register* primitive instantiation. Line 11 illustrates register input and output definition, line 12 displays register clock connection, line 13 defines clock sensitivity and line 14 depicts enable signal connection.

```
1  class FD1QSX010(Dataflow):
2      def __init__(self, *args, **kwargs):
3          super(FD1QSX010, self).__init__(*args, **kwargs)
4          self.CP = InPort(ObjProps=BitVec(1))
5          self.D = InPort(ObjProps=BitVec(1))
6          self.Q = OutPort()
7          self.TE = InPort(ObjProps=BitVec(1))
8          self.TI = InPort(ObjProps=BitVec(1))
9
10         self.In = (self.D & ~self.TE) | (self.TI & self.TE)
11         reg = Register(In = self.In, Out = self.Q)
12         Clk = reg.Clk.connect(self.CP)
13         SensClk = reg.setClockSensitivity("RisingEdge")
14         reg.En.connect(Literal(1))
```

Listing 5.3: ToD of a library cell

Similar procedures are repeated until all combinational and sequential cells in a library cell are converted into their respective ToD classes. A python ToD file named *lib_cells.py* is generated that is named as library ToD.

Netlist-ToD generation

A gate-level netlist provides a representation of the logical operations performed by a circuit or design. It showcases the circuit's structural composition using a combination of logic gates, which includes both simple gates and complex cells sourced from the standard cell library. Verilog RTL is converted into a netlist via a synthesis tool, e.g., Yosys. After the netlist is generated, it is translated into a netlist-ToD, in a similar fashion to the Library-ToD translation. The

Verific netlist parser is utilized to read and extract netlist information such as: (i) module name, (ii) ports name, size and direction, (iii) wires, (iv) module's instantiations and connections, (v) instances of the module, and (vi) function of the module.

Listing 5.4 illustrates a snippet of a netlist of an ALU component named *ALU_HMINUS*. The component contains two wires (lines 2-3), three pins (lines 7-9) and two instantiated modules (lines 10-14, lines 15-19).

```
1  module ALU_HWMINUS(Outp, In00, In01);
2    wire _000_;
3    wire _001_;
4
5    /* rest of the code */
6
7    input [31:0] In00;
8    input [31:0] In01;
9    output [31:0] Outp;
10   EO2X010 _120_ (
11     .A(In00[0]),
12     .B(In01[0]),
13     .Z(Outp[0])
14   );
15   OR2IX010 _121_ (
16     .A(In01[0]),
17     .B(In00[0]),
18     .Z(_118_)
19   );
20   /* rest of the code */
```

Listing 5.4: Netlist example

The corresponding MetaRTL ToD of the previous netlist snippet is shown in Listing 5.5. Initially the generated library ToD is imported (line 1) and the module's name is converted into the corresponding class name. This class is then established in a structural format, as depicted in line 3. Ports, their direction and their sizes are then defined in lines 6-8. Lines 10-11 describe how the wires are defined in the ToD via *Conn()* connection object. Lines 13-22 display the instantiation of two modules *comp0* and *comp1* respectively. The module *comp0* is an instance of *EO2X010* cell from the library ToD as shown in line 13. After that, all pins of the cell are properly connected as displayed in lines 15-17. *INDEX* is a MetaRTL primitive that indexes certain defined bits of signal, and as an example, in line 15 it is indexing bit 0 of *In01* input.

A similar process is utilized to convert all modules of the netlist into the ToD constructs and in the end the instantation of all modules is performed. A python ToD file named *netlist.py* is generated that is named as netlist Tod. The netlist ToD imports the generated library ToD, and using MetaRTL framework, the MoD of the netlist is generated. The next step of the flow consists of creating a fine-grained MoD.

```
1  import lib_cells as cells
2
3  class ALU_HWMINUS(Structure):
4      def __init__(self, *args, **kwargs):
5          super(HWMINUS_004, self).__init__(*args, **kwargs)
6          self.In00 = InPort(ObjProps=BitVec(32))
7          self.In01 = InPort(ObjProps=BitVec(32))
```

```
8         self.Outp = OutPort(ObjProps=BitVec(32))
9
10        self._000_ = Conn()
11        self._001_ = Conn()
12
13        comp0 = cells.EO2X010(parent=self)
14        self.insComponent(comp0)
15        comp0.A.connect(INDEX(self.In00 , Literal(0)))
16        comp0.B.connect(INDEX(self.In01 , Literal(0)))
17        comp0.Z.connect(self.Outp0)
18        comp1 = cells.OR2IX010(parent=self)
19        self.insComponent(comp1)
20        comp1.A.connect(INDEX(self.In01 , Literal(0)))
21        comp1.B.connect(INDEX(self.In00 , Literal(0)))
22        comp1.Z.connect(self._118_)
```

Listing 5.5: Netlist-ToD example

5.3.2 Fine-Grained MoD

The process of converting a gate-level netlist into a mixed-granularity model is essential and involves several steps that work together. First, the netlist is translated into a MetaRTL-based gate-level model. Then, this model is combined with the original model to create a mixed-granularity model. Throughout this process, specific model transformation operators are used to ensure each step is completed accurately.

To illustrate the application of these operators, consider a scenario where fault injection campaigns are targeted at the Execute stage of a processor core. The original MoD of the processor describes the entire processor, while the netlist MoD only covers a subcomponent of the processor, specifically the Execute stage, at the gate-level granularity. In this situation, the *locate* operator identifies and isolates the Execute stage component in the original model, preserving its attributes such as ports, connections, and literals. Once the component has been located, the *replace* operator replaces it with the corresponding component from the netlist model, at the module level, ensuring that input and output ports are aligned. This replacement process generates a new and refined model that describes the DUT in fine-grained gate-level granularity, while maintaining the original format for all other elements. The end result is a mixed-granularity model that more accurately depicts the design of the processor. A high-level illustration of this transformation process can be observed in Figure 5.4.

5.4 Fault Injection through Model Transformation

The principal aim of the fault simulation flow developed on this thesis is to architect an automated procedure for the seamless integration of fault injector modules into mixed granularity RTL Models. This procedure involves the construction of a specialized circuit or module dedicated to the injection of specific fault models, also known as fault injectors. Following this purpose, a fault-collapsing methodology is created to systematically eliminate equivalent or redundant faults. Finally, model transformation is utilized to integrate the fault injectors into the mixed granularity models.

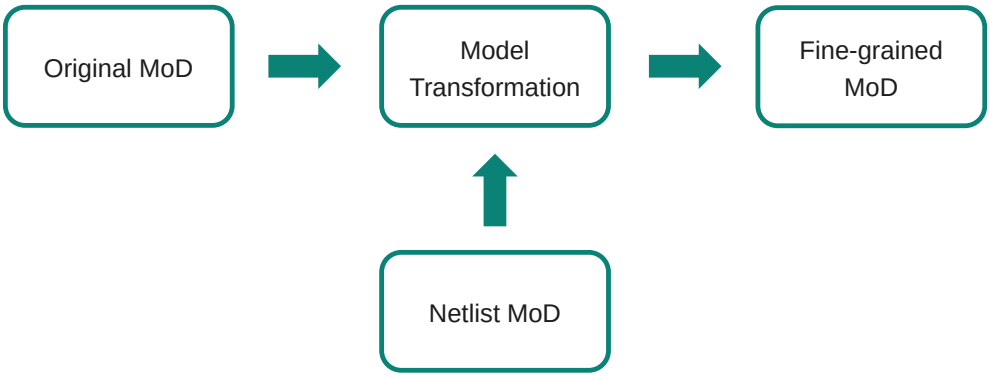


Figure 5.4: Creation of a fine-grained MoD

5.4.1 Fault Injectors

In the context of this thesis, *saboteurs* are used as fault injectors. A saboteur represents a distinct and specialized element deliberately integrated into the design that enables altering the value of a design signal. When the saboteur is activated, it modifies the value of designated signals equipped with saboteurs. A dormant saboteur does not exhibit any modification to the design's behavior. The saboteur's operational states, i.e., active and dormant, are governed by a control signal. Notably, the impact of saboteurs is restricted only to the ports/connections associated with the components of the design. This characteristic makes usage of saboteurs exclusive to structural descriptions.

Figure 5.5 displays the fault injectors that are utilized on the fault simulation framework. The *IN* signal represents the precise location for fault injection within the design. The control signal *CTRL*, on the other hand, serves as the fault injector activator and selects the necessary fault model to inject. To remain dormant, i.e., to not inject any fault such that $OUT = IN$, the *CTRL* signal of the fault injector depicted in Figure 5.5a should have a value of **"100"**. Otherwise different fault models can be injected such as: (i) stuck-at-0 when *CTRL* is equal to **"000"**, (ii) stuck-at-1 when *CTRL* is equal to **"X10"** (X is dont'care value), and (iii) bit-flip when *CTRL* is equal to **"101"**.

The fault injector displayed in Figure 5.5b extends the previous injector to inject timing faults too. Timing faults, also known as transition-delay faults, emerge from inherent manufacturing defects and produce a delayed reaction within the system. In this context, specific parts of the design compute the accurate outcome, but the response occurs at a later point in time compared to the fault-free scenario [74, 103]. To implement this scenario, a multiplexer and a register are added to the fault injector to delay the response by a clock cycle when the control signal activates the timing fault. Precisely, when the least significant bit of control signal *CTRL* is set to 1, *Out* signal's value is similar to *IN* signal's value but delayed by one clock cycle. when the least significant bit of control signal *CTRL* is set to 0, fault injector behaves similar to the saboteur presented in Figure 5.5a.

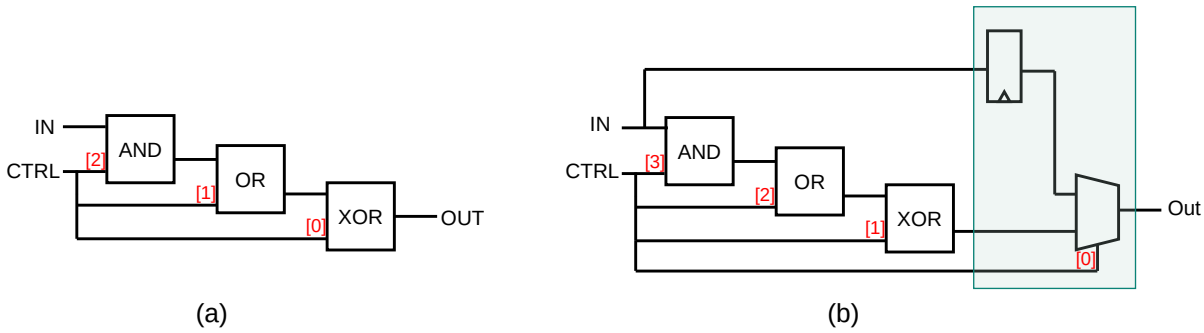


Figure 5.5: Fault injectors

5.4.2 Fault Collapsing

As discussed in Chapter 3, fault collapsing offers the capability to reduce the count of faults within a given circuit. The collapsed fault list provides advantages in processes like fault simulation, fault diagnosis, test pattern generation, and similar tasks, by conserving time and computational resources [128]. In the fault simulation framework, fault collapsing is applied to reduce the count of faults within the gate-level design.

Specifically, fault collapsing is applied on library cells by creating a fault matrix for each cell. Then, based on the fault matrix equivalent faults are removed. The complete fault collapsing process is automated. A Python script is developed that reads the library cells and automatically generates a fault matrix for all the cells. This fault matrix is afterwards employed in the saboteurs' insertion process to determine the appropriate fault injection module for each pin of the library cell.

Figure 5.6: Collapsed fault injectors [79]

For example, the fault injector of Figure 5.6a can inject only stuck-at-0 and bit-flip fault model while the fault injector of Figure 5.6b injects stuck-at-1 and bit-flip fault model. The shorter path of the fault injectors can further improve simulation performance and provide smaller area to keep pace with the FPGA resources.

5.4.3 Insertion of Fault injectors

The fault injectors are inserted at every connection of the fine-granular MoD through model transformation. The transformation process is composed of three consecutive steps:

1. **Location of the target component**: To begin the search for the target component within the MoD, the *locate* operation commences at the highest-level module, tracing through

the modules until the target component is found. Once found, all module details of the library cells within the component are located and saved for future transformation stages. This includes capturing all connections and ports associated with each library cell.

2. **Adding fault injectors**: In this step, the library cell and its connections are navigated using the *add* transformation operator. These library cells can either be pure logic components comprised of Boolean gates or they can be sequential cells such as registers or latches. There are a few rules to follow while inserting the fault injectors such as:
   - a fault injector should be inserted at every pin of a combinational cell,
   - a fault injector should not be inserted at a clock pin of a sequential cell,
   - a fault injector should be inserted only once at a connection signal,
   - and a fault injector should be inserted only at the driver of a fan-out.

3. **Replace**: In the final step of fault injection insertion, all the old connections of the library cells are replaced with new connections from the fault injectors.

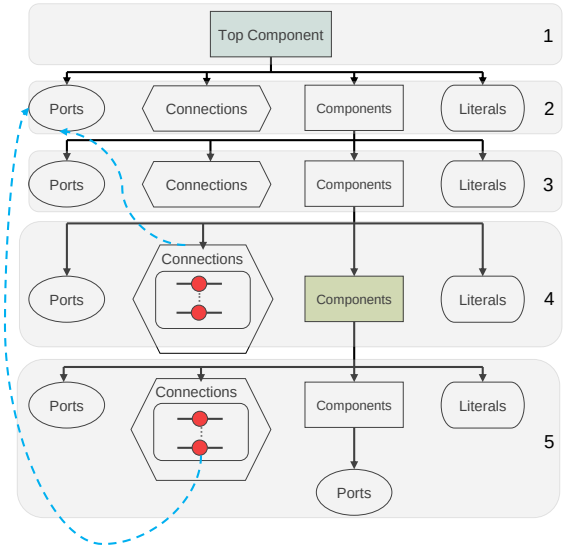


Figure 5.7: Example of inserting fault injectors into the MoD

The previous procedure is reiterated for all library cells within the designated component, adhering to the specified constraints. Upon completion of this phase, each cell is equipped with an fault injector. For illustration purposes, a basic example of an MoD with five hierarchical levels can be used to demonstrate this concept as shown in Figure 5.7. Each component within the MoD can be associated with ports, connections between them, sub-components, and literals (constant values). The target of a fault injection campaign is the component highlighted in green. As a result, all of its connections (nets) and sub-module (e.g., library cells) connections will be transformed. For simplification, fault injectors are depicted as red circles and nets as straight lines. As shown in the picture, all of the previous connections are replaced each with a new connection that contains a saboteur. Each saboteur has a fault control line added as a primary input for the top-level hierarchy, while the framework automatically propagates its object properties. This allows for high controllability and observability for the fault injection process.

5.5 Equivalence Checking and Property Checking

Fault simulation on mixed granularity RTL requires two consecutive transformation processes: (i) transformation of the gate-level netlist into a fine-grained MoD, and (ii) transformation of fine-grained Mod into a fine-grained MoD equipped with fault injectors. The transformations are fully automated and utilize Metagen build-in APIs. Nevertheless, if a bug occurs during the transformation process, it has the potential to introduce faults or errors in the resulting transformed model [28]. To address this issue, an automated formal verification flow has been developed as illustrated in Figure 5.8.

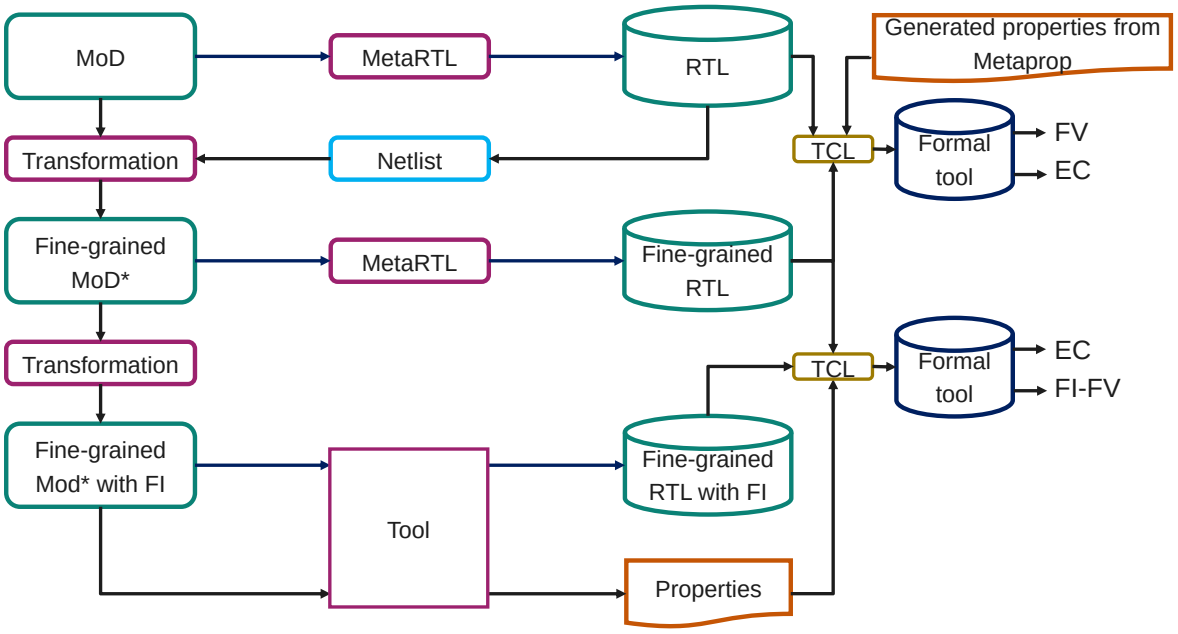


Figure 5.8: Formal verification flow of mixed granularity RTL

As previously described in this chapter, the initial step involves leveraging the MetaRTL flow to generate the RTL that aligns with the design model MoD. Subsequently, formal verification (FV) is executed employing the generated properties generated from Metaprop. The purpose of formal verification is to verify whether the design conforms to the specified requirements, thereby avoiding scenarios in which functional bugs manifest as faults during fault simulation.

In the primary transformation phase, the netlist is transformed into a fine-grained MoD*. This process is facilitated through the MetaRTL flow, and to ensure the absence of bugs or errors during this transformation, an Equivalence Check is conducted between the original RTL and the fine-grained RTL. The Equivalence Check is carried out automatically using TCL scripts that activate the formal tool.

Subsequently, the fine-grained MoD is equipped with fault injection capabilities by inserting fault injectors. An intermediate Python reader is employed to extract information from the intermediate fault injection model MoFI, thereby generating constraints for an Equivalence Check between the non-faulty fine-grained RTL and its counterpart with inserted fault injectors. These constraints are essential for the formal tool to restrict fault injectors to not inject

any faults, thereby mimicking the behavior of the original RTL. Furthermore, the Python reader generates properties necessary for the fault injectors formal verification (FI-FV). These properties serve to verify whether the fault injectors indeed inject the intended fault models; for instance, if a fault injector's control signal is set to inject a stuck-at-0 fault model, the output of the fault injector should indeed correspond to a stuck-at-0 condition.

The automated formal verification workflow provides a thorough evaluation to ensure that there are no bugs present in the transformations. This creates a strong basis for fault simulation on mixed-granularity models, allowing for complete confidence in the accuracy of the results.

Chapter 6

Model-Driven FPGA-Based Fault Emulation

Simulation-based fault injection is widely employed to assess the effects of faults and evaluate the resilience of designs. Fault simulation stands out due to its cost-effectiveness, complete observability, and capacity to encompass various fault models without incurring additional overhead. The fault simulation process can be realized through the introduction of saboteurs and mutants into the gate-level/RTL code or model, or via the utilization of special simulators. While fault simulation presents numerous advantages, it often requires prolonged runtime durations to execute comprehensive fault campaigns. The fault simulation on mixed granularity models presented in Chapter 5, aims to overcome this major runtime drawback by performing fault simulation on mixed RTL/gate-level granularity. However, it is important to acknowledge that even with this approach, there remain limitations associated with the overall simulation speed, which typically operates at a few kHz.

Fault emulation techniques have emerged as an approach to facilitate fault injection campaigns. These techniques improve the overall productivity by reducing the time needed for fault injection while simultaneously maintaining the capacity to deliver thorough evaluations. Emulation operates at significantly higher frequencies in comparison to commercial simulators, resulting in significant accelerated fault injection runtimes and a major improvement is observed while running longer input sequences. In this thesis, an automated framework for fault emulation has been developed, which combines enhanced observability and controllability of injected faults with notable performance improvements. A well-known limitation of emulation-based methods for conducting fault campaigns is the insufficient availability of I/O ports. To counter this, an innovative design architecture is presented. The proposed novel fault emulation framework significantly reduces the manual efforts and can be scaled to emulate an entire CPU subsystem on an FPGA with LUTs. The emulation-based fault injection framework, i.e., fault emulation framework, builds upon the model-driven RTL generation framework, MetaRTL. The framework is fully automated, extends and deploys the existing fault injection framework by providing a novel architecture for fault emulation using FPGAs. The fault handling meta-model, presented in Chapter 4, specifies the necessary architecture by describing various fault injection campaigns. The fault emulator architecture is composed of four main components such as: (i) Fault Controller, (ii) Design-under-Test (DUT), (iii) Postprocessing block, and (iv) Data harvesting logic. Furthermore, the framework also incorporates an *on-the-fly* technique

for the analysis of fault emulation, effectively mitigating potential FPGA memory bottlenecks.

This chapter provides a thorough description of the architecture underlying model-based fault emulation. It commences with a holistic view of the entire architecture, followed by detailed explanation of individual components. An outline of the proposed flow was previously published in [76].

6.1 Overview of the Fault Emulator Architecture

The fault emulator framework serves as an extension of the preceding fault injection flow outlined in Chapter 5. The fault injection flow automatically inserts fault injectors into the design model MoD and propagates fault control lines as primary inputs (PIs) of the design. The subsequent step involves the utilization of MetaRTL to generate the design, i.e., the DUT, equipped with fault injectors that are added to specific components for fault injection purposes. The fault emulator modifies the transformed MoD by adding extra components to enable automated fault injection and analysis, as depicted in Figure 6.1. All the additional components are described using the ToD and they are configurable adhering to the fault handling metamodel that is detailed in Figure 4.2.

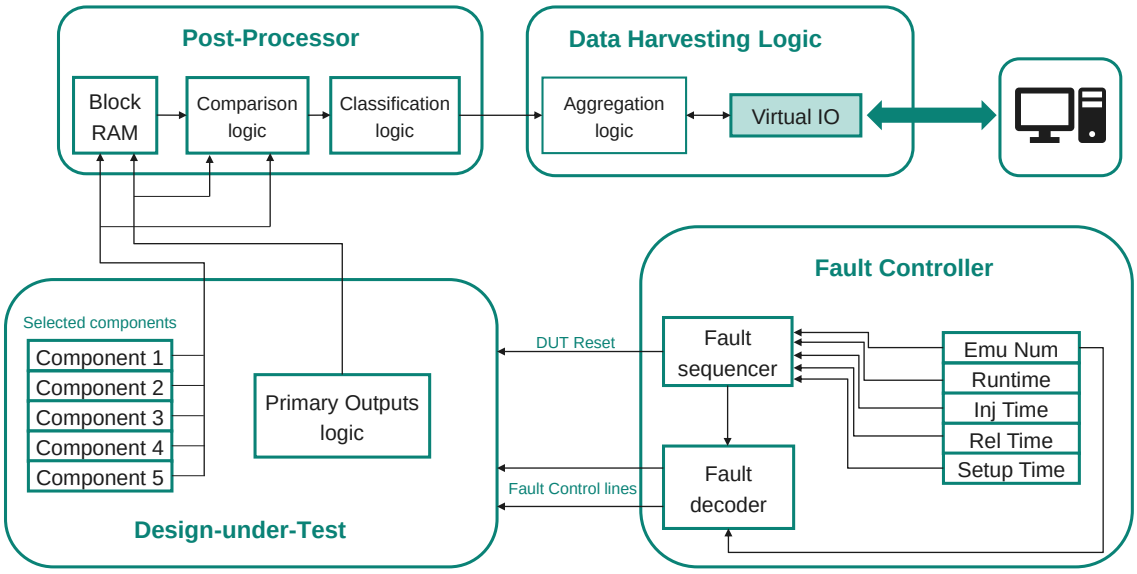


Figure 6.1: Architecture of fault emulator [76]

The *Fault Controller* operates by driving the fault control lines, and setting up the injection and the release of individual faults into the DUT with precise time points. The controller maintains a generic nature, adaptable to different fault injection campaigns as specified in the fault handling metamodel, such as Statistical Fault Injection or Exhaustive Fault Injection. For precise temporal fault management, the controller requires multiple parameters including emulation number, runtime, injection time, release time, and setup time.

The *Post-Processor* component captures and preserves emulation traces, encompassing primary outputs (POs) as well as specific data from selected components. This trace data is stored

within the Block RAM for both the fault-free and faulty operational states. The user can designate the desired outputs for analysis through the Fault Analyzer section of the fault handling metamodel. By comparing the traces of the chosen signals, the framework then categorizes the faults according to their effects.

The *Data Harvesting Logic* component plays a pivotal roles in the system by receiving fault classification values generated within the Post-Processor block. Subsequently, it transmits these classification values to the Host PC for further assessment, thus ensuring a seamless and automated flow of fault analysis from the emulation environment.

A comprehensive and detailed description of the aforementioned components follows.

6.2 Fault Controller

The central component for fault injection is the *Fault Controller*, which directly manipulates the fault control lines connected to the PIs of the DUT to inject the faults. The Fault Controller is created based on the information from the metamodel. Its hardware is generated using the typical model-driven approach of MetaRTL, therefore enabling adjustable configurations according to different fault handling parameters. Figure 6.2 displays the block schematic of the Fault Controller. The Fault Controller module operates independently without the need for external control inputs.

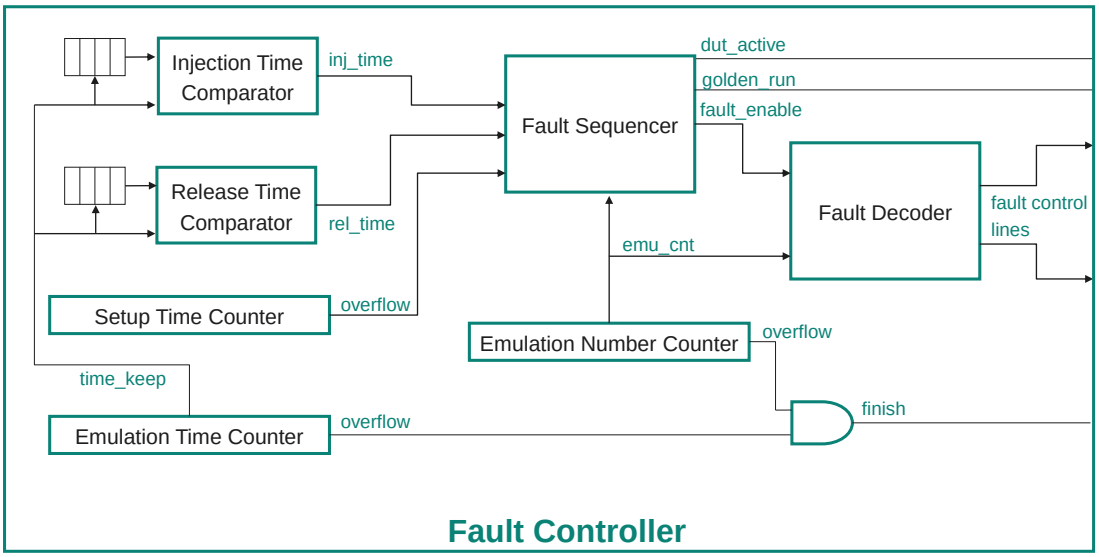


Figure 6.2: Fault controller [76]

As can be seen from Figure 6.2, precise fault injection at specific time points is supported by distinct time-keeping counters. Given that multiple sequential emulations are necessary for injecting numerous faults, the *Emulation Number Counter* records the iteration count of each emulation run. The *Emulation Time Counter* computes the clock cycles in each emulation run, while the *Setup Time Counter* identifies the moment when the emulation should start, i.e., when the reset is deactivated. The *Release Time Comparator* and *Injection Time Comparator* are two comparators tasked with determining the times for fault injection and release. These

times are defined by the fault handler model (outlined in Chapter 4) and are stored in internal Block RAMs (BRAMs) within the FPGA-based emulator. The comparators compare the values stored in the RAM against the *Emulation Time Counter* value (*time_keep*) and raise two flags *inj_time* and *rel_time* whenever the required clock cycle for fault injection/release is reached. The *Fault Sequencer* within the Fault Controller controls the fault injection process by reading the data from the counters and comparators, communicating with the *Fault Decoder* to alter the relevant fault control lines' states according to the decoded fault. Additionally, a *finish* signal indicates the end of the fault injection campaign accordingly by checking the overflow values of *Emulation Number Counter* and *Emulation Time Counter*.

6.2.1 Fault Sequencer

The Fault Sequencer module, predominantly employed for time-keeping purposes, is constructed through the utilization of a FSM, as depicted in Figure 6.3. Initially, this module deactivates all fault control lines and initiates a golden emulation run without any injected faults, denoted by the *golden_run* signal (Figure 6.2).

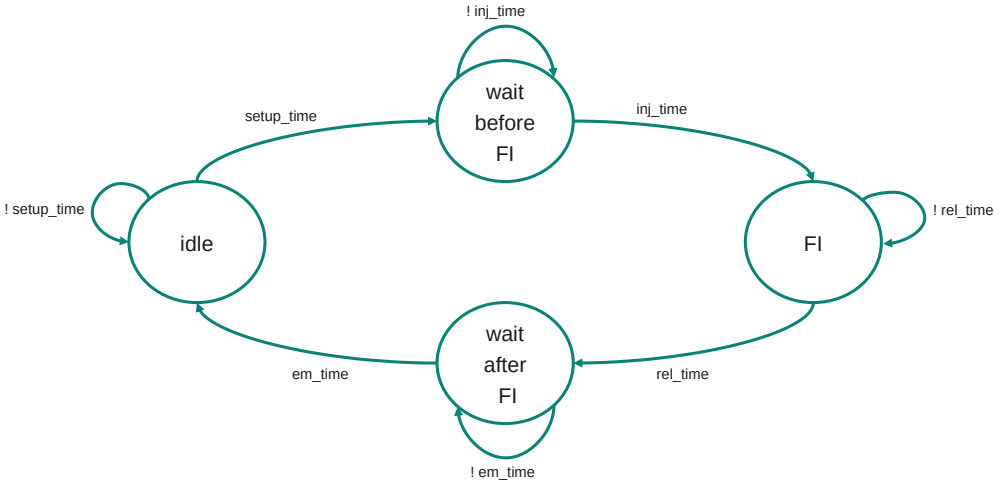


Figure 6.3: Fault Sequencer FSM

During the fault injection process, DUT starts at the **idle** state, and it remains in this state until the *setup_time* duration elapses. After this time is finished, the DUT goes to a wait state **wait before FI**. In this state, the DUT is released from the reset state, the emulaton process initiates and no fault is injected. The process remains within this state until the predetermined injection time is reached, as detected by the *Injection Time Comparator*. Subsequently, after the *inj_time* flag is activated, the process advances to **FI** state, triggering the assertion of *fault_enable* signal to the Fault Decoder. This action results in the injection of the specific fault in question, which is maintained until the release time is achieved, detected through the *Release Time Comparator*. The remaining portion of the process unfolds in the **wait after FI** state, persisting until the *Emulation Time Counter* overflows. This fault injection process concludes by transitioning back to the idle state and waits again for the next sequence of faults. The subsequent faults are injected in a sequential manner by traversing the same state sequence.

6.2.2 Fault Decoder

The Fault Decoder is used to map or decode the value of the *Emulation Number Counter* into appropriate values for fault control lines according to the fault handling model. Internally, it consists of individual transcoder blocks which drive one fault control line each, as shown in Figure 6.4.

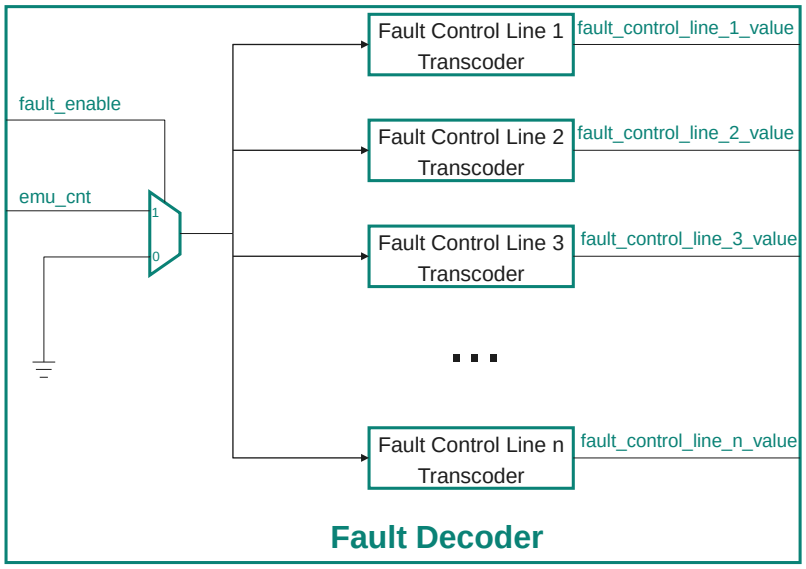


Figure 6.4: Fault Decoder

The transcoders are created with minimal effort by using built-in MetaRTL transcoder primitive. The *fault_enable* signal, originating from the Fault Controller, controls the propagation of the incoming *emu_cnt* value sourced from the *Emulation Number Counter* towards the transcoder modules. This process is commanded via a simple 2:1 multiplexer. The transcoder decodes the *emu_cnt* value, and when its value aligns with the decoding conditions, corresponding fault lines are driven. Conversely, for all other circumstances, the transcoder drives fault-free control values as its output.

For instance, consider a scenario where it is intended to inject stuck-at-0 and stuck-at-1 faults into the DUT at the first control line. This is achieved by enabling *Fault Control Line 1* during the first and second emulation runs, respectively. Consequently, the transcoder would activate this control line whenever the *emu_cnt* assumes the values of 1 and 2. Typically, the fault handling model also determines the emulation number linked to specific fault locations, thus facilitating the creation of transcoders.

6.3 Postprocessing Block

After a fault is injected into the DUT, it is necessary to analyze its effects on the design's behavior. The *Postprocessing block* plays a pivotal role to analyze and classify the faults according to their effect. A block level overview of the Postprocessor is displayed in Figure 6.5. The *Block Memory* is a central element of the module and is built via a FPGA vendor-specific BRAM. During the golden run, all POs traces are stored into the BRAM via a demultiplexing logic that

uses *golden_run* signal as a select input. The same signal concurrently serves as the *write_en* input for the BRAM, guaranteeing that write operations are exclusively confined to the golden emulation phase. The *BRAM Address Counter* utilizes clock cycles as distinct address points for writing data into the BRAM, e.g., POs traces for clock cycle 3 are stored in address location $0x03$. It is neccessary to note that this data is written only when the DUT is active.

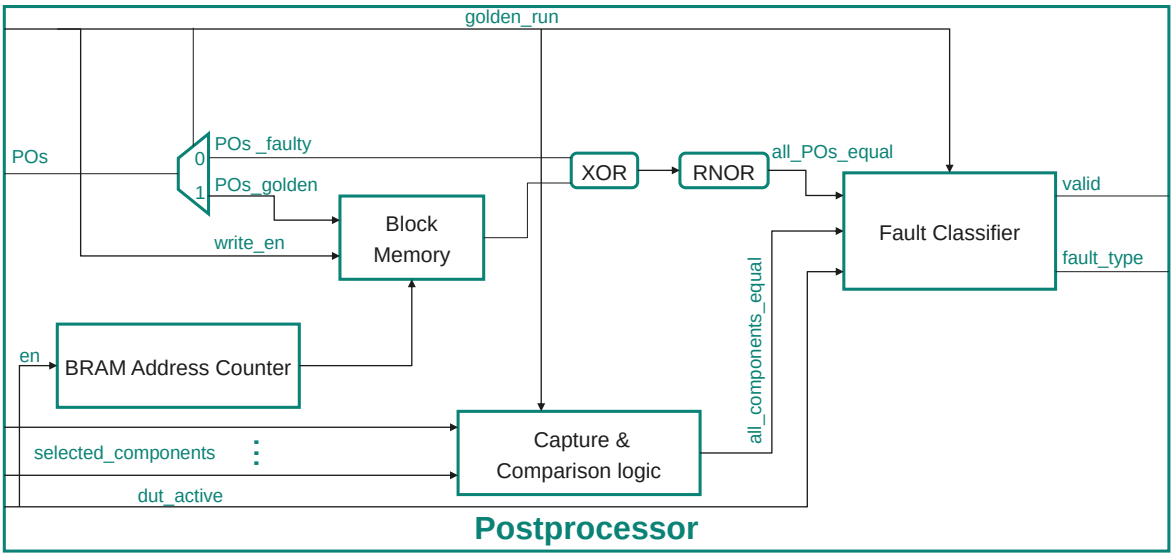


Figure 6.5: Postprocessor [76]

The comparison mechanism is carried out by directing the extracted golden values (retrieved from the BRAM) and the corresponding values for the fault-injected emulation through a bit-wise XOR gate. If the POs value during the fault injection remain similar to the non-faulty one, then an all-zero bit vector will be produced by the XOR gate; otherwise, it generates a non-zero value. The Reduced-NOR (RNOR) gate produces a logical '1' when all input bits are 0. The resulting output from the RNOR gate manifests as the *all_POs_equal* signal that is employed in fault classification. Similarly, *Capture & Comparison logic* module follows the same principle and compares internal registers of selected components and generates the flag signal *all_components_equal*. The registers of selected components are defined in the fault handling model.

In the fault classification stage, the *Fault Classifier* module takes center stage. Its functionality is governed by a FSM, which takes input signals including *all_POs_equal*, *all_regs_equal*, and other control signals. Depending from high or low states of these signals, the *Fault Classifier* classifies faults into the various categories such as: (i) silent faults when both *all_POs_equal* and *all_regs_equal* are high, (ii) latent faults when *all_POs_equal* is high but all_regs_equal is low, and failures when both *all_POs_equal* and *all_regs_equal* are low. The outcome of this classification is encoded in the *fault_type* signal. When a fault is successfully classified, the *valid* signal is triggered, facilitating the storage of the classified fault value in upcoming processes for *Data harvesting logic*.

6.4 Data Harvesting Logic

In the context of fault emulation campaigns, a conventional practice involves preserving the categorized fault values within the FPGA for the entire duration of the campaign. After the fault injection campaign has finished, these classified values are transmitted back to the host PC [102]. Following a similar approach, this thesis explores a strategy centered on the Xilinx Virtual-IO IP. Here, the results of fault classification are captured and stored within array structures embedded within the FPGA itself. After the injection campaign concludes, these accumulated values are retrieved from the host PC. However, a noteworthy constraint within the Virtual-IO platform pertains to the time-related overhead incurred by each read/write operation. This overhead could extend to approximately a minute, varying from different dependencies. To address this limitation, a mitigation strategy involves aggregating the fault classified values into more extended bit-vectors. Subsequently, these values are retrieved in batches, thereby optimizing the readout process. An overview of this *Data Harvesting Logic* is illustrated in Figure 6.6.

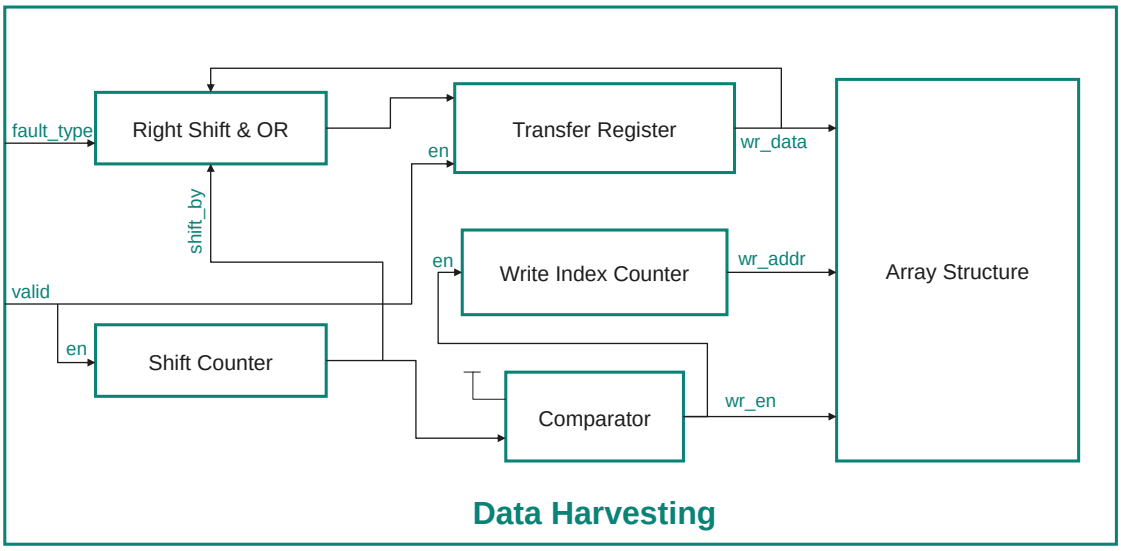


Figure 6.6: Data Harvesting Logic [76]

The *Transfer Register* is utilized to capture and store the classified fault values and its contents are subsequently transferred to a unified *Array Structure* once the register reaches its capacity. The *Array Structure* has the same bitwidth as *Transfer Register* (256 bits), while its depth is determined according to the Fault Handling model. The *Data Harvesting Logic* takes as input *valid* and *fault_type* signals from the *PostProcessor* module. A logical right shift operation is performed on the *fault_type* value according to the *Shift Counter* output data. On each new fault classification, this module increments the output *shift_by* with the size of each individual fault value such that it does not overwrite the existing data. After shifting, a logical OR is performed with the existing data of *Transfer Register*. The outcome of the operation is then stored again in the register.

Upon reaching the maximum shift value as determined by the *Comparator*, this module triggers the writing of the Transfer Register contents into the Array Structure by activating

wr_en signal. Afterwards, the process restarts, with the *Transfer Register* being refilled with new classified fault values. The *Write Index Counter* controls the write positions within the *Array Structure*, incrementing by 1 with each overflow of the *Shift Counter*. By the end of the emulation campaign, the classification details for all injected faults are stored within the *Array Structure*. This data is then transferred from the FPGA to the host PC.

6.5 Fault Emulation Optimizations

The addition of configurable features to the framework requires specific alterations to the design of its components. These changes are necessary to ensure flexibility and customization within the framework and allow for optimizations to its architecture, memory requirements, and fault emulation duration. Further details on the adjustments made to the framework and their implications are provided in the following subsections.

6.5.1 Memory Optimization

Generally, fault-tolerant computing systems employ the Lockstep approach [13] to execute identical operations in parallel. The redundancy inherent in lockstep systems enables both error detection and error correction as detailed in Chapter 3. Fault emulation campaigns can run for very long periods of time, thus creating a bottleneck of BRAM utilization. Thus, to avoid this constraint, the original fault emulation framework (see Figure 6.1) is modified in a similar fashion to the Lockstep approach as shown in Figure 6.7. The modified version incorporates an additional instance of the DUT. The two DUT instances are named as: (i) *Fault-Injection DUT (FI-DUT)* that is equipped with fault injectors, and (ii) *Golden DUT* that has no fault injectors and produces fault-free design behavior.

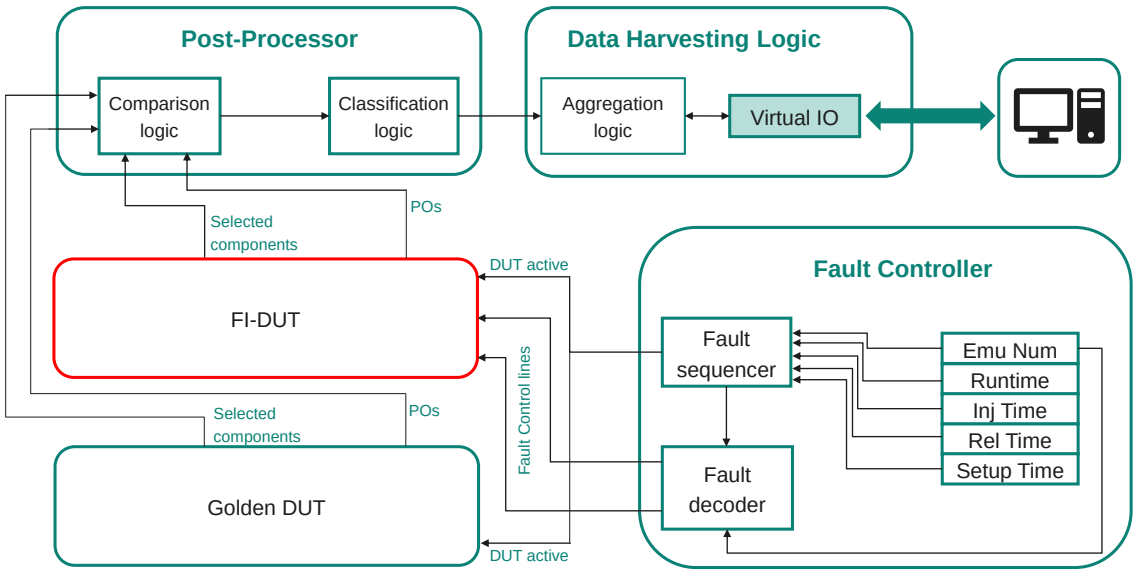


Figure 6.7: "On-the-fly" fault emulation architecture [76]

This approach relies on the concurrent operation of both the *Golden-DUT* and the *FI-DUT*.

Both instances are driven with identical parameters throughout each fault emulation iteration, enabling real-time comparison of their POs and register values for classification purposes, i.e., comparison "on-the-fly". This novel strategy removes the necessity of a dedicated golden emulation process to store fault-free values in the BRAM, as these values are acquired directly from the Golden-DUT. This refinement optimizes the overall workflow significantly. Furthermore, a notable benefit of this strategy is its capability to extend emulation runtimes considerably. This advantage comes from the elimination of BRAM resource bottlenecks, as there is no necessity to store traces within the BRAM.

6.5.2 Time Optimization

In the process of fault analysis of safety-critical systems, it is of utmost significance to observe faults that lead to failures. Any time a failure has been detected, immediate measures must be taken to handle the failure. Hence, in the context of fault propagation analysis, it becomes feasible to pause the emulation precisely at the point of detection of the faults that cause failures. This approach eliminates the need to continue the emulation until its predefined completion. Consequently, this strategy can lead to substantial savings in terms of clock cycles, particularly when conducting successive fault injections involving lengthy instruction sequences. To enable this efficient approach, a signal named *flush* is created, connecting the *Postprocessor*'s Fault Classification Logic to the Fault Controller. Consequently, the emulation corresponding to the particular fault (that causes failure) is stopped, enabling the system to move rapidly to the next fault.

Chapter 7

Safety Verification of Hardened Processor Cores

Despite the diversity of fault injection methods that have been developed over time, fault simulation and emulation remain prevalent choices due to their cost-effectiveness and simplicity in implementation. However, a notable drawback of fault simulation and emulation lies in their constrained fault coverage, as it can only identify faults that have been explicitly modeled. Moreover, the effectiveness of fault sensitization is dependent on the input stimuli, which might result in certain faults remaining undetected, that can lead to potential design failures. ATPG remains the most common technique to produce exhaustive input test patterns, encompassing all feasible design scenarios to uncover faults. While ATPG excels in testing combinational designs, it encounters challenges when dealing with more complex sequential designs. Moreover, there remains a difficulty in automatically deriving and applying functional constraints for testing purposes. This challenge arises because ATPG methods predominantly focus on the structural aspects of the circuit, primarily its gate-level representation.

ISO26262 recommends formal methods for verifying the integrity of the most safety-critical elements. In this thesis, a novel approach is introduced that combines model-driven approach with formal techniques to systematically analyze the impact of faults in processor designs. The process starts by utilizing the model-driven strategy to create designs with a mixed granularity. Then, a scalable formal verification technique for processors is proposed, enabling the verification of design hardening mechanisms and delivering fault analysis outcomes without additional efforts.

The technique enables the verification of all error correction and detection mechanisms in the presence of faults without any additional manual effort and without any white-box design knowledge. The existing flow of generating mixed granularity models enables gate-level fault modeling only on intended design components, thus reducing the complexity of formal methods.

Following the model-driven approach, the technique presented in this Chapter is entirely automated, demanding only minimal human intervention, predominantly focused on parameter configuration. The fault analysis process synergizes with the design generation flow, providing valuable insights into redundant generated modules and/or sub-modules. In the subsequent sections, first an overview of the background that enables the proposed technique is presented. Next, the exhaustive fault injection on the processor core is described. Finally, the details of the

formal fault propagation analysis methodology are outlined.

7.1 Background

In this section, some fundamental techniques are visited, such as Safety Transformation Flow and Complete-Symbolic State Quick Error Detection (C-S^{2}QED). These revisions are included for the sake of self-containment within this thesis.

7.1.1 Safety Transformation Flow

Model transformation is an important part of model-driven engineering due to its numerous advantages. As detailed in preceding chapters of this thesis, model transformation has been utilized to facilitate fault injection into the design through the insertion of fault injectors, often termed as saboteurs. The process of model transformation is further utilized for hardening designs as well as presented by [30, 64]. ISO26262 recommends various levels of risk classification (ASIL), thus various safety mechanisms should be used according to the safety requirements. Clearly, the varying nature of safety mechanisms requires large development efforts to create safety-critical designs. To combat this issue, functional safety features are automatically added within the MoD layer through model transformation. Safety constraints are applied to the original MoD to obtain a transformed one with various safety mechanisms against various faults.

The transformation strategy starts with a base MoD and reconfigures its flip-flops in accordance with specified safety constraints, thereby producing a hardened MoD. The safety mechanisms constraints and features are defined in the Safety Transformation metamodel illustrated in Figure 7.1. The root class is named as *SafetyConstraints* and has a one to many relationship with *SafetyGroup* class. For each safety group, there exists a corresponding safety mechanism defined by *SafetyPattern*. Currently, the support safety mechanisms include DMR, TMR, Error Detection Code (EDC) using Parity (PAR), and ECC utilizing Hamming code for SEC and extended Hamming code for supplementary SEC-DED [64]. The *ModuleName* defines the top level module of the design that is subject of hardening while *Path* further specifies the exact location of the component. The designated flip-flops to harden are identified within the *FlipFlop* class. When protecting specific bitfields of a flip-flop is required, these can be selected within the *BitRange* class.

Once the safety constraints have been set as desired, a Python script will locate the so-called safety groups in the base MoD and the selected flip-flops and bitfields inside them. For each of the latter, a wrapper is created containing a copy of it with the additional safety mechanism. The original flip-flop is then deleted and replaced by the corresponding wrapper. After every safety constraint has been considered, the output is a hardened MoD that can be integrated into the RTL generation flow to generate RTL code of safety-critical designs [64].

7.1.2 Complete Functional Verification of Processor Cores

In the domain of processor verification, the conventional practice has predominantly relied on simulation-based methodologies, utilizing a mix of random and constrained simulation patterns.

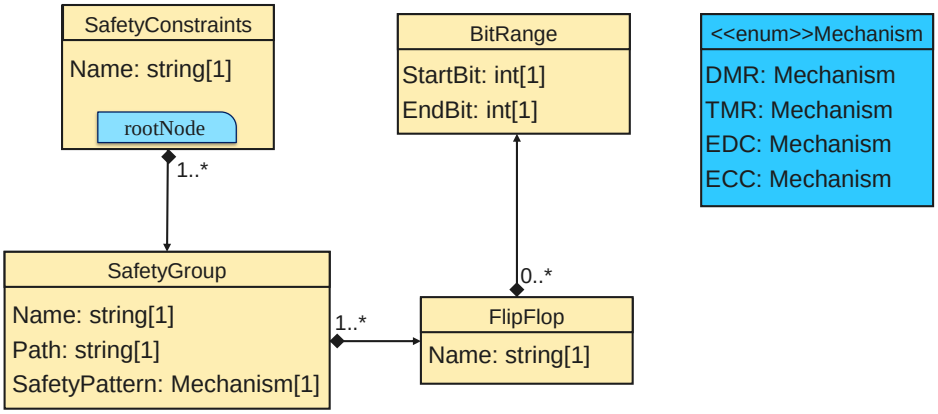


Figure 7.1: Safety Transformation metamodel [64]

However, a novel technique introduced by Fadiheh et al. [59] called Symbolic State Quick Error Detection (S^2QED) has emerged for formal processor verification. This method is especially focused at detecting hard-to-find bugs or confirming their absence.

The basis of the S^2QED methodology relies on proving that every instruction functions independently from its program context. In order to accomplish this goal, the verification model uses two identical processor cores. These cores are constrained to fetch and execute the same instruction at an arbitrary point in time, represented as t. Moreover, the model unrolls processor instances across a time window corresponding to the instruction's execution duration. For illustrative purposes, let us consider a hypothetical scenario involving a 5-stage pipelined processor encompassing Fetch (IF), Decode (ID), Execute (EX), Memory-Access (MEM), and Write-back (WB) stages. Figure 7.2 illustrates the conceptual framework of the verification model designed for the aforementioned processor. This model comprises two distinct CPUs, both of which fetch the same Instruction-Under-Verification (IUV). It is essential to note that CPU_1 initiates its operation from a clean state, meaning that its pipeline is flushed. Conversely, CPU_2 starts its execution from a symbolic state, thereby allowing the formal verification tool to consider diverse scenarios of legal program contexts. It is worth noting that the S^2QED approach assumes that both CPU instances are supposed to uphold QED consistency, i.e., containing matching values within their corresponding Register Files, even after the Write-Back process. Consequently, any logic bug would lead to a state of QED inconsistency [59].

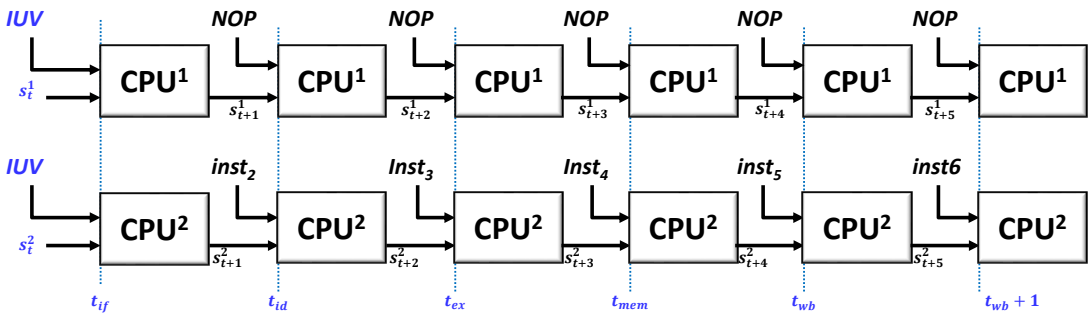


Figure 7.2: S^2QED verification model for a 5-stage pipelined CPU

Figure 7.3 illustrates the application of a property to the S^{2}QED model using an Interval Temporal Logic (ITL) syntax. It's important to note that both CPU instances fetch the same instruction. In this scenario, CPU$_1$ initiates in a clean state, signifying that solely No-Operation (NOP) instructions are fetched prior to the IUV. Both instances maintain a QED-consistent state prior to the final write-back of results. The property checks that the CPUs will remain in a QED-consistent state even after the Write-Back stage. In the event of any logic bugs, a QED inconsistent state would consequently arise [59].

assume:
  at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
  during $[t_{IF}+1, t_{WB}]$: *cpu1_fetched_instr*() = NOP;
  at t_{WB}: *qed_consistent_registers*();
prove:
  at $t_{WB}+1$: *qed_consistent_registers*();

Figure 7.3: S^{2}QED property

The S^{2}QED technique is proficient at detecting multiple instruction bugs originating from sequences of instructions. However, it falls short in detecting single instruction bugs that could produce similar effects on both instances, thereby not resulting in an inconsistent state. For instance, an error in the Decoder could misinterpret an "add" instruction as a "subtract" instruction, causing both cores to exhibit analogous behavior. To overcome this drawback, Devarajegowda et al. introduced an enhanced version of S^{2}QED named C-S^{2}QED [51]. C-S^{2}QED offers a comprehensive solution by addressing both single and multiple instruction bugs. This approach is based on a series of extended C-S^{2}QED properties, each focusing on a distinct class of instructions, such as register-type, immediate-type, memory-type, and more. Essentially, for each instruction class, a specific C-S^{2}QED property is generated by making slight adjustments to the original property and introducing additional assumptions. These modifications constrain the instruction class at the Decode stage according to its type and ensure the CPU is prepared for the next instruction. Additionally, an extra step is implemented to verify if the values written to CPU$_1$'s Register File conform to specifications, e.g., proving that the content in the destination register aligns with the sum of the source registers' contents for an "add" instruction. Figure 7.4 shows an example of the C-S^{2}QED property for immediate type instructions.

7.1.3 RISC-V CPU Metamodel

All C-S^{2}QED properties have been generated using MetaProp and the design RTL has been generated using MetaRTL. Considering the complexity and various features that a CPU has, it is necessary to formalize its specifications via the metamodel. Figure 7.5 depicts the metamodel that captures a RISC-based CPU specifications. The root class of the metamodel is called MetaRISC and the metamodel itself is composed of four elements that describe the CPU behavior such as [47]:

- **Architectural States**: A processor includes key architectural state elements like the general-purpose register file (GPR), program counter (PC), and control and status registers (CSR). To depict these elements, a composition relationship is established from

assume:
at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
during $[t_{IF}+1, t_{WB}]$: *cpu1_fetched_instr*() = NOP;
at t_{ID}: *ready_for_next_instruction*();
at t_{ID}: *imm_type_instr*();
at t_{WB}: *qed_consistent_registers*();
prove:
at $t_{WB}+1$: *qed_consistent_registers*();
at $t_{WB}+1$: *cpu1_reg_value*(reg_addr @ t_{ID}) = *expected_value*(funct_type @ t_{ID})

Figure 7.4: C-S^{2}QED property for immediate type instructions

the root node to the *ObjectProperties* class. This class has distinct attributes for defining different properties of the state elements. For every architectural state element, an *ObjectProperties* instance is created containing different attributes.

- **Instructions**: This element captures the instructions and their extensions within an ISA. This is realized through a composition connection from the root node to the *Instruction* and *Extension* classes. Every instruction is associated with a distinct extension in the ISA. Each instruction bears attributes like *Name*, *Mnemonic*, and an *Active*, denoting its support within the microarchitecture. An instruction's impact on the CPU's architectural state is outlined as a series of modifications it introduces. For each affected state element, an *Instruction* instance has an associated *InstructionBehavior* component. To encapsulate the activities executed by the instructions, the *InstructionBehavior* class is required.
  Encoding Tree: This element constructs a hierarchical structure to define an instruction word's configuration and parameters. Through *RangeNode* and *Opt* classes, specific bit-positions and their corresponding values are determined recursively, creating a tree to represent the instruction word's format. Additionally, the root node contains parameters and their encodings that define the bit positions for operands within the instruction word.
  Exceptions: This element captures the behavior of exception events in a way similar to the instructions. Upon encountering an exception event, the processor initiates a sequence of actions, updating state elements prior to executing a predetermined exception routine.

The metamodel serves as a key element also when performing safety verification of the processor as detailed in the following.

7.2 Overview of Safety Verification of Processor Cores

A diverse array of hardware-based hardening methods, encompassing ECC, CRC, TMR, DMR, Lockstep, as well as Parity codes, can be strategically integrated into designs. This integration plays a pivotal role in protecting the integrity and reliability of safety-critical components. The design hardening approach, as detailed above, transforms the design model by adding various safety mechanisms. Consequently, a heightened level of effort becomes necessary for the verification of two critical aspects:

1. The functionality of the hardening techniques: This involves confirming whether the im-

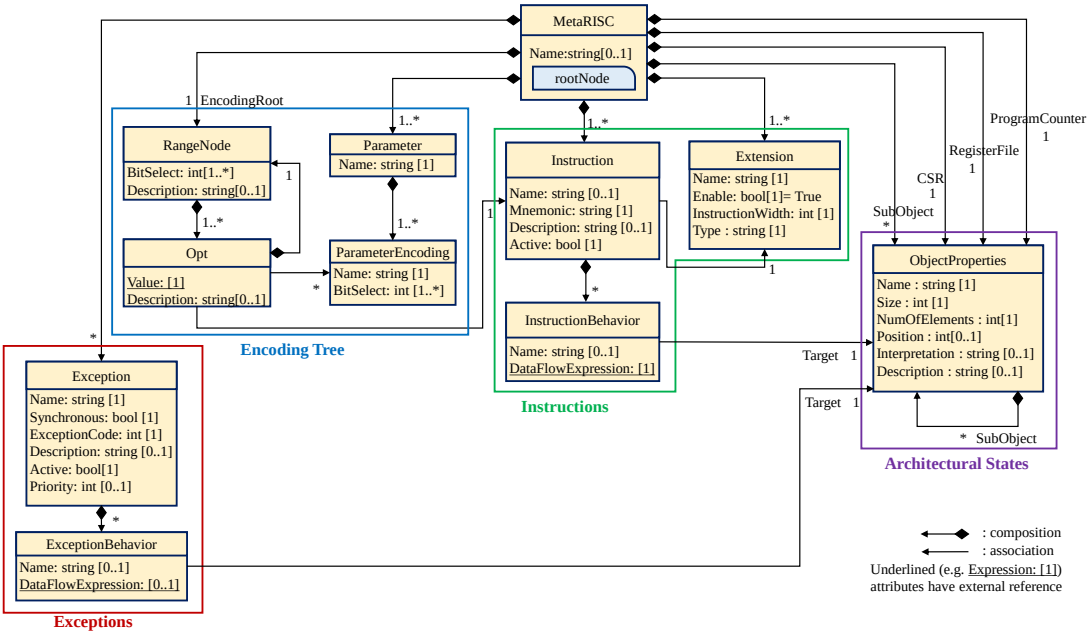


Figure 7.5: CPU metamodel [47]

plemented fault detection and correction mechanisms behave as intended.

2. Seamless system integration: This process requires careful examination to guarantee that the safety mechanisms are properly integrated into the system without breaking its intended functionality.

The verification process for ensuring the correct operation and proper integration of these additional hardware modules entails an increase of effort, time, and a deep knowledge of the intricate design details. In this thesis, an automated safety verification flow is developed that verifies hardened processor cores as depicted in Figure 7.6. The main benefit of the flow relies on removing the requirement for these extra verification tasks for hardened designs, resulting in a reduction of both time and effort.

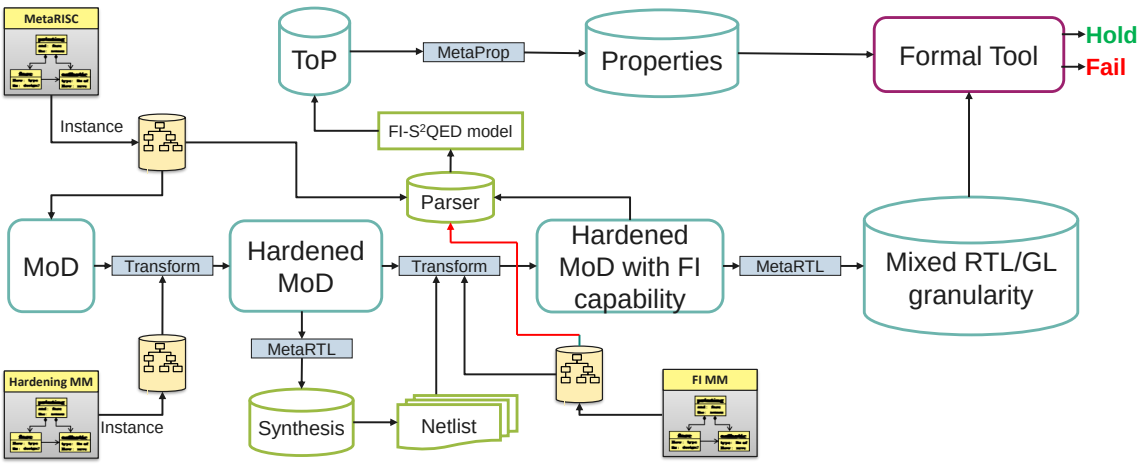


Figure 7.6: Processor safety verification flow

In the initial stage, the design model MoD is generated adhering to the MetaRISC metamodel (see Figure 7.5), which outlines the precise specifications of the processor. Then the Safety Transformation flow is applied to the MoD, by transforming it into a Hardened MoD that contains various safety mechanisms protecting selected components.

Subsequently, the automated process of generating mixed granularity models is executed, as elaborated in Chapter 5. This flow involves generating the RTL representation of the hardened MoD, performing synthesis to produce a gate-level netlist, converting the netlist back into an MoD, and then further transforming the MoD by incorporating fault injectors.

Within this context, a parser extracts important information from both the original MetaRISC instance and the transformed MoD. This extracted information serves as the basis for creating the computation model for safety verification, known as *Fault-Injection S^2QED (FI-S^2QED)*. The FI-S^2QED model is specifically designed to include essential constraints for facilitating fault injection. Concurrently, the parser identifies and extracts the precise fault model from the fault handling metamodel instance, e.g a single fault model or a multiple fault model.

Lastly, the ToP employs the FI-S^2QED verification model, and the MetaProp tool is used to generate formal properties. The generated properties combined with the generated design in a mixed granularity are then subjected to verification using a formal verification tool. This verification process proves if the properties hold true or fail, thus determining whether hardened designs conform to the specifications.

7.3 Exhaustive Processor Fault Injection

In this section, a detailed explanation of the the safety verification computation model FI-S^2QED is given including the corresponding fault model to inject.

7.3.1 Verification Computation Model

The FI-S^2QED computation model is an extension of the S^2QED approach, illustrated in Figure 7.2. In this model, two distinct yet identical processor cores are constrained to fetch the same instruction. However, in the FI-S^2QED model, both processors have the capability of introducing faults. However, CPU_1 is designated as the clean-state CPU and is constrained to remain fault-free.

The creation of the FI-S^2QED model involves a preliminary step where the processor undergoes formal verification using C-S^2QED. This verification ensures that any anomalies or failures triggered by faults are not mistakenly identified as functional defects. To offer a visual representation of the generated FI-S2QED property, refer to Figure 7.7. As illustrated in the figure, there are additions made to the property to incorporate dependencies or constraints that remain applicable at any time point. Specifically, CPU_1 is under the constraint of not introducing any faults, whereas CPU_2 is governed by a predefined fault model. The activation of a fault is accomplished through the utilization of a Boolean predicate, *cpu2_inject_fault*, employed as an ITL macro. This macro signifies the user-defined fault model that is to be applied. Faults in a system can affect both data flow and control flow aspects of the program. Therefore extra consistency checks are required on critical registers like the PC. These checks are denoted in green within the property representation.

dependencies:
  constraint: *cpu1_no_fault*();
  constraint: *cpu2_inject_fault*();
assume:
  at t_{IF}: *cpu2_fetched_instr*() = *cpu1_fetched_instr*();
  during $[t_{IF}+1, t_{WB}]$: *cpu1_fetched_instr*() = NOP;
  at t_{EX}: *qed_consistent_PC*();
  at t_{WB}: *qed_consistent_registers*();
prove:
  at $t_{EX}+1$: *qed_consistent_PC*();
  at $t_{WB}+1$: *qed_consistent_registers*();

Figure 7.7: FI-S^{2}QED property

To fix injected faults, an error correction mechanism should be present, thus the QED consistency check should continue to hold even post-fault injection. It's essential that the chosen fault model aligns with the characteristics of the safety mechanisms integrated within the system e.g., if the safety mechanism is able to detect or correct only a single fault, then a single fault should be injected.

To illustrate the working of the property, let us consider the hardening of the ID-EX pipeline register using SEC codes. Within this context, the formal tool has the capability to inject a fault into the ID-EX pipeline register of CPU_2. This is achieved by steering the FI_Control_lines in accordance with the defined single fault model(*cpu2_inject_fault* macro). The underlying premise is that if the safety mechanism is prone to functional and integration bugs, the impact of faults will be visible. If no bugs are present in the hardened design, the CPU's behavior will align with that of a fault-free one, thus the QED consistency check will continue to hold. It is important to note that the property's focus is not on the specific type of hardening mechanism employed, but rather on evaluating whether the CPU complies with the specifications even in the presence of faults. Remarkably, only minimal "white-box" knowledge is necessary, primarily the understanding of the number of bits to be corrected. If the hardening mechanism includes error detection, a slight modification to the property is required. Instead of exclusively verifying *qed_consistent_registers()*, the property should validate a Boolean expression: *qed consistent registers() or fault detected()*. In cases where the fault has not caused any impact (whether it's been corrected or has remained silent), the *qed consistent registers()* condition would remain valid. However, should the fault propagate to the Register File, the consistency check would fail. Yet, if the safety mechanism successfully detects the fault (expressed via the *fault detected()* macro, serving as the output of the safety mechanism), the property would still stand true. This is due to the property's nature of verifying a Boolean expression of "or" type.

7.3.2 Fault Model Definition

The macro *cpu2_inject_fault* defines certain constraints to inject a fault at CPU_2. Algorithm 6 displays the pseudocode utilized to generate the macro via Metaprop.

When activated, *FI_Control* signals drive faults into various locations. Since these signals

Algorithm 6 Pseudocode for defining FI-S^{2}QED fault model

Input: FI_control_list, nr_injected_faults
Output: Fault model

```
1:  function STABLE_FAULT_CTRL(FI_control_list)
2:      stable_fc = 1
3:      for FI_ctrl in FI_control_list do
4:          stable_fc = stable_fc   AND FI_ctrl = past(FI_ctrl)
5:          return stable_fc
6:      end for
7:  end function
8:
9:  function CONCATENATE_FAULT_CTRLS(FI_control_list)
10:     concatenate_list = ∅
11:     for FI_Ctrl in FI_control_list do
12:         concatenate_list.add( FI_Ctrl)
13:     end for
14:     return CONCAT(concatenate_list)
15: end function
16:
17: function FAULT_MODEL_INJECT(nr_injected_faults, FI_control_list)
18:     fault_ctrls = CONCATENATE_FAULT_CTRLS(FI_control_list)
19:     return $COUNTONES(fault_ctrls) = nr_injected_faults
20: end function
21:
22: function CPU2_INJECT_FAULT(nr_injected_faults, FI_control_list)
23:     return FAULT_MODEL_INJECT() AND STABLE_FAULT_CTRL()
24: end function
```

are propagated as primary inputs of the design, it is easy to control the fault model for injection. The formal tool can drive the *FI_Control* signals with any valid legal value at any time point *t*; thus, it is essential to constrain these signals to inject only a specific fault at all time points. For example, if the tool randomly drives the control signal to inject a stuck-at-0, it should not change the fault type thereafter. To achieve this, the *STABLE_FAULT_CTRL* procedure (lines 1-7) generates the *stable_fc* constraint that forces all control signals to remain stable, meaning their value should be equal to the previous time point value (line 4). The subsequent phase encompasses the *CONCATENATE_FAULT_ CTRLS()* procedure (outlined in lines 9-15). This procedure concatenates all fault control signals by employing the *CONCAT* operation (line 14). The *CONCAT* operation, a primitive operation of MetaProp, takes a list as input and concatenates the corresponding signals. The *FAULT_MODEL_INJECT* procedure (lines 17-20) proceeds to insert the fault(s) by manipulating specific values within the concatenated signal. This is accomplished utilizing the *COUNTONES()* function, which identifies the number of bits with a '1' value in a signal. Line 19 introduces an assumption that controls the injection of a predefined number of faults (denoted as *nr_injected_faults*). This implies that only "n" bits within the concatenated signal will bear a '1' value, while the remaining bits will be set to '0'. To illustrate, if the intention is to inject two faults, the concatenated signal will have only two '1' bits, with the rest being '0'. Consequently, the formal tool can inject a predetermined count of faults by activating the *FI_Control* signals. Lastly, the concluding procedure (lines 22-24) yields the *cpu2_inject_fault* macro. This macro is expressed as a boolean AND operation, incorporating the assumptions in line 23.

The FI-S^{2}QED property facilitates the injection of a specified count of faults across all potential design locations. This features allows for a comprehensive verification of hardened safety-critical processor cores in the presence of faults.

7.4 Formal-Based Fault Propagation Analysis

Within a digital design, potential fault locations can span from thousands to even millions. However, it's essential to recognize that not all faults result in failure scenarios, as certain faults might not propagate to primary state registers or primary outputs. These particular faults are labeled as redundant faults, as their presence does not disrupt the circuit's behavior. For a holistic assessment of all plausible faults within the design, it is necessary to create suitable input test patterns. These patterns sensitize the faults and evaluate fault coverage. In the following, a novel formal-based fault propagation analysis approach will be discussed. This approach stems from the fundamental principles of the FI-S^{2}QED property.

7.4.1 Verification Computation Model

The primary concept behind formal-based fault propagation analysis is to produce a set of k-FI-S^{2}QED properties, where the value of *k* corresponds to the number of locations in the design where a fault can occur. Each individual property is designed to introduce a fault at a specific location. The principle of this approach lies in evaluating whether these properties adhere to the FI-S^{2}QED model's criteria. Should any property fail to meet these criteria, it signifies that there exists a sequential set of instructions capable of sensitizing and propagating the fault. In

this context, the fault is classified as detected. Given the substantial number of potential fault locations, the process of generating these properties requires an automated approach. For this purpose, MetaProp serves as a crucial tool, enabling the generation of these properties with minimal effort. This involves making only minimal modifications to the original FI-S^{2}QED model, thus ensuring an efficient and comprehensive fault propagation analysis.

Illustrated in Figure 7.8 are multiple FI-S^{2}QED properties, i.e., k properties. Each individual property is designed to introduce a fault at a distinct location within CPU". These properties share a common characteristic in terms of their fault injection mechanisms, differing only in the macro, *CPU2_FAULT_INJECT* employed for injecting the fault. Notably, this macro's operation involves enabling a single *FI_Control* signal while simultaneously deactivating all other fault control signals. This configuration ensures that only the specified fault is injected while other potential fault sources remain inactive.

```
for (int k=1; k ≤ total_fault_locations; k++) {
 property FI-S²QED_k is :
   dependencies:
     constraint:                cpu1_no_fault();
     constraint:                cpu2_inject_fault(k);
   assume:
     at t_IF:                   cpu2_fetched_instr() = cpu1_fetched_instr();
     during [t_IF + 1, t_WB]:   cpu1_fetched_instr() = NOP;
     at t_EX:                   qed_consistent_PC();
     at t_WB:                   qed_consistent_registers();
   prove:
     at t_EX + 1:               qed_consistent_PC();
     at t_WB + 1:               qed_consistent_registers();
}
```

Figure 7.8: k-FI-S^{2}QED properties

To elaborate, let us consider a modest-sized design featuring only 10 distinct fault locations. Employing MetaProp, a collection of 10 distinct FI-S^{2}QED properties is generated, denoted as {FI-S^{2}QED$_1$, FI-S^{2}QED$_2$, ..., FI-S^{2}QED$_{10}$}. Each property is devised to introduce a single fault, e.g., the FI-S^{2}QED$_1$ property only activates $FI_Control_1$ (with the formal tool assuming any allowed value), while concurrently imposing restrictions on all other fault control signals, forbidding them from injecting any faults: {FI_Control$_2$=0, FI_Control$_3$=0, ..., FI_Control$_{10}$=0}.

Notably, any property that results in a failure signifies that the associated fault has been detected, while properties that hold imply redundant faults. This approach involves the generation of k-FI-S^{2}QED properties, a process similar to the one depicted in Figure 7.6, but without consideration for hardened designs. This systematic approach provides a high level of confidence in fault propagation analysis through an exhaustive functional testing approach. Furthermore, it can be perceived as a technique for optimizing design area, effectively finding redundant design signals and emphasizing functional redundancy over structural redundancy.

7.4.2 Fault Model Definition

The pseudocode outlining the definition of macro *CPU2_FAULT_INJECT* is illustrated in Algorithm 7. The *CPU2_FAULT_INJECT* procedure is designed to take the index k as an input parameter. Within this procedure, all fault control signals except the one linked to the given index are deactivated (lines 3-5). The outcome of this procedure is the logical AND operation performed on all the deactivated fault control signals. Clearly, the macro CPU2_FAULT_INJECT does not constrain the corresponding *FI_Control* signals. This approach allows the formal tool to inject a fault by presuming any valid value for the given control signal.

Algorithm 7 Pseudocode for defining k-FI-S^{2}QED fault model

Input: k
Output: Fault model

```
1: function CPU2_FAULT_INJECT(k)
2:     deactivated_fc_signals = ∅
3:     for FI_ctrl in FI_control_list do
4:         if FI_Ctrl != FI_control_list[k] then
5:             deactivated_fc_signals.add (FI_Ctrl.deactivate())
6:         end if
7:     end for
8:     return LAND(deactivated_fc_signals)
9: end function
```

Chapter 8

An Automated and Effective Approach for SBST Generation Targeting RISC-V CPUs

The continuous scaling and intricate manufacturing processes involved in digital designs may lead to various faults in the Integrated Circuit (IC) output. Over the years, numerous DFT techniques have been developed to ensure comprehensive and efficient testing of ICs during the manufacturing process.

To enhance the testing of SoC structures, particularly those with complex processor cores, additional test circuitry is incorporated into the design. As discussed in Chapter 7, testing sequential designs poses challenges for ATPG techniques. Therefore, the insertion of scan chains becomes necessary to enable effective testing. Scan chains facilitate the transfer of test data throughout the design, allowing for the testing of internal circuitry. By connecting internal design registers/flip-flops via scan chains, observation and control of inaccessible internal design nodes become possible. This approach ensures that only the combinational logic between registers/flip-flops needs testing.

Following the integration of scan chains into the design, testing is conducted using test vectors, typically stored in Automatic Test Equipments (ATEs). Although the added circuitry enhances testability and fault coverage, the larger size of required test vectors results in longer test times and higher ATE costs [90]. Another prevalent DFT infrastructure is Built-in Self Test (BIST), which incorporates a self-contained test circuit within the design. BIST techniques are commonly categorized as Memory-BIST (MBIST) for testing memory circuits and Logic-BIST (LBIST) for testing logical circuits.

The implementation of various DFT infrastructures introduces significant overhead in terms of both area and performance. In response to the challenges posed by testing processor cores, an alternative technique has emerged known as Software-based Self Test (SBST). This method involves the testing of processor cores through instructions, providing a viable alternative to hardware-centric solutions like BIST. Unlike hardware-based self-test, which requires the activation of the non-functional BIST mode, SBST can be conducted during the normal operational mode of the processor without requiring design modifications or the incorporation of additional hardware structures such as ATEs or scan chains [39]. The software program, comprising a set of instructions, engages the processor or system, generating test patterns and analyzing the outcomes to identify defects or faults [39].

The primary challenge encountered by most SBST approaches lies in generating adequate

test patterns capable of exercising a wide range of faults. To address this challenge, various test generation techniques are employed. Psarakis et al. [111] classifies different SBST techniques into four major categories:

- **Functional**: These techniques rely only on the ISA without requiring extensive testing knowledge, as demonstrated in [121, 109]. While applicable to any design, functional SBSTs suffer from low fault coverage due to their lack of structural information.
- **Structural, hierarchical with precomputed stimuli**: These approaches utilize formal verification methods to generate deterministic test patterns, as seen in [134, 113, 60]. While achieving high fault coverage, structural SBSTs based on formal methods may encounter scalability issues with complex designs.
- **Structural, hierarchical using constrained test generation**: These techniques leverage simulation-based learning to generate suitable patterns, e.g., [39]. The primary advantage is reduced test complexity, but the quality of testing relies on the accuracy of simulation models.
- **Structural, RTL**: These techniques can independently generate test programs regardless of processor complexity, as demonstrated in [92, 90, 66]. Structural, RTL SBST is effective for testing complex processor architectures, though it requires in-depth knowledge of the architecture and exhibits limited automation.

Clearly, all categories of SBST techniques encounter shared challenges encompassing: (i) achieving a high fault coverage, (ii) reducing the complexity of test generation, (iii) scalability to complex designs, and (iv) enhancing automation. This thesis introduces an automated and efficient approach for generating SBST tailored for RISC-V processor cores, addressing all aforementioned challenges. The proposed methodology falls within the category of *Structural, hierarchical with precomputed stimuli*, relying on formal verification techniques such as assertion-based property verification. The extraction of deterministic test patterns through formal properties results in a high fault coverage, effectively addressing challenge (i). The integration of formal verification with fault simulation reduces the number of required properties by eliminating faults detected by the same test pattern, thus resolving challenge (ii). Moreover, the versatility of the technique has been demonstrated through its successful application to various components of a RISC-V CPU, establishing full scalability and overcoming challenge (iii). Notably, the approach is fully automated, providing a solution to challenge (iv). In addition, the SBST generation is extended with a novel PFC technique, ensuring a high fault detection rate aligned with different ASILs from the ISO 26262 standard. Consequently, the proposed approach not only facilitates test generation but concurrently delivers high fault detection while maintaining full automation.

Subsequent sections of this chapter provide an overview of the SBST flow, followed by a detailed explanation of the test pattern generation technique. The chapter concludes by describing the PFC and its configuration for fault detection.

8.1 Overview of the SBST

Generally, the SBST generation is computationally expensive, tedious and prone to errors. To combat these drawbacks, the SBST generation flow presented in this thesis, combines and utilizes extensively automation frameworks such as Metagen, MetaRTL, MetaProp and MetaFI

(see previous chapters for more details on the frameworks). The SBST generation flow is combined of two major tasks: (i) Test Pattern Generation and (ii) PFC.

Figure 8.1 illustrates a high level overview of the complete SBST generation flow. The flow is composed of several steps as following:

1. The DUT, i.e., the CPU, undergoes the fault injection transformation flow. The component that is object to testing is kept on a gate-level granularity while the rest of the design remains in its original RTL. During this transformation process, the fault list is automatically generated.
2. After transforming the DUT in FI-DUT, a *miter* circuit is automatically created consisting of the original design and the one with fault injection capabilities. A miter circuit is a common term in verification and design domain that describes two identical designs that are being compared while driving them with identical set of inputs.
3. The next step consists of injecting faults via formal properties and checking whether the fault has any effect on the design. The main idea is to write a single property that checks that the outputs of the miter circuit are identical in the presence of the fault, and if they are not identical, then a counterexample is produced by the formal tool.
4. A script extracts the inputs values from the counterexample thus creating a pattern which is able to sensitize and propagate the fault to the primary outputs/registers. The pattern is stored in a log file in a hexadecimal (hex) format. This format facilitates setting input values in the simulation testbench.
5. After creating the pattern for a single fault, fault simulation is performed for all other faults using the same pattern as stimuli. If any other fault is sensitized and propagated via the same pattern, then this fault is dropped and not considered anymore in the pattern generation via the formal properties. Formal properties are applied again for the rest of the faults until all of them have been detected by a test pattern. If some fault does not produce any counterexample, i.e., it does not have any effect in the design, it is considered undetectable.
6. As a next step, the individual test patterns are concatenated to create a complete test program. It is necessary to note that the CPU is reset via a custom CSR instruction at the beginning of the individual test pattern such that there is no dependencies between different patterns.
7. Two custom PFC CSR instruction, i.e., *PFC Start* and *PFC End* are added to the complete test program. The *PFC Start* instruction initializes the PFC hardware module to start hashing of the instructions and *PFC End* stops the hashing of instructions. The hashed value is stored into the internal PFC CSR.
8. As a last step, the value stored in the PFC CSR is compared with the expected hashed value of the instructions which has been precomputed. If there is any mismatch, a fault has been detected.

8.2 Test Pattern Generation

Test Pattern Generation (TPG) is one of the major tasks of the SBST generation flow. This task is computationally expensive and time-consuming due to several steps that are required to generate a complete pattern to test all stuck-at faults in the design. The main idea of the TPG

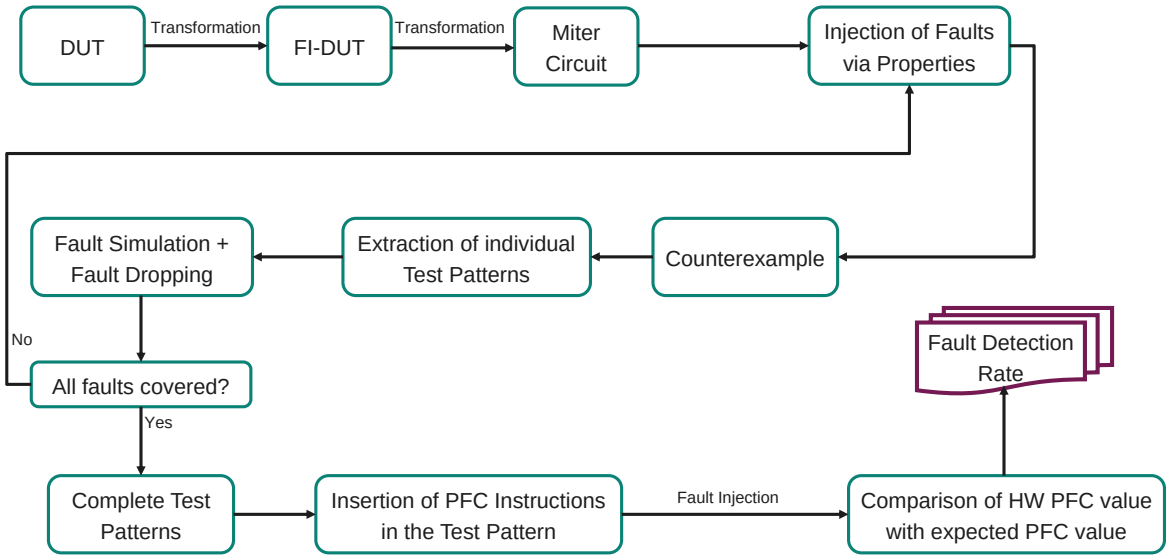


Figure 8.1: SBST generation flow

presented in this thesis is using formal properties to check whether a fault affects the DUT (the CPU), and if yes, providing the input sequence, i.e., instructions, that sensitize and propagate fault effects in the primary outputs/registers of the design. The formal properties consider a *miter circuit* and the formal tool would provide a counterexample if the designs of the miter circuit are not equivalent in the presence of a fault. The counterexample provides the sequence of instructions that are able to test the fault, therefore the test pattern can be extracted from the counterexample.

8.2.1 DUT and Properties

Figure 8.2 illustrates the complete setup of the DUT and the formal properties that are used to verify fault effects. The setup is fully automated using various generation frameworks such as MetaRTL, MetaProp and MetaFI. Furthermore, various Tcl scripts are utilized to automate the access of the formal tool and the execution of the properties. Initially, the DUT, i.e., the RISC-V based CPU, is generated via MetaRTL. Clearly, the primary element of the RTL generation flow is the RISC-V CPU metamodel that defines the CPU specifications. Some of the main characteristics of RISC-V architecture are the extensability and customizability. Based on this feature, a custom CSR instruction has been added that is able to reset the Register File of the CPU. This instruction is added at the beginning of each test pattern to avoid data dependencies between different individual test patterns, therefore each test pattern starts from a clean state. For clarification, since the formal properties start from a reset state, the input sequence starts from a clean CPU state too. Therefore, it is necessary to reset the Register File to concatenate several individual test patterns. As an example, the complete pattern set (CTP) would look like the following: CTP = {custom CSR, TP_1, custom CSR, TP_2,..., custom CSR, TP_n }, where n represent the number of individual test patterns (TP).

After the CPU has been generated, it undergoes the process of the Fault Injection Transformation as explained in Chapter 5 where the desired components of CPU (defined in MetaFI

metamodel) are equipped with fault injection capabilities. A miter circuit is automatically created that is composed of the original DUT and the DUT with fault injection capabilities (FI-DUT).

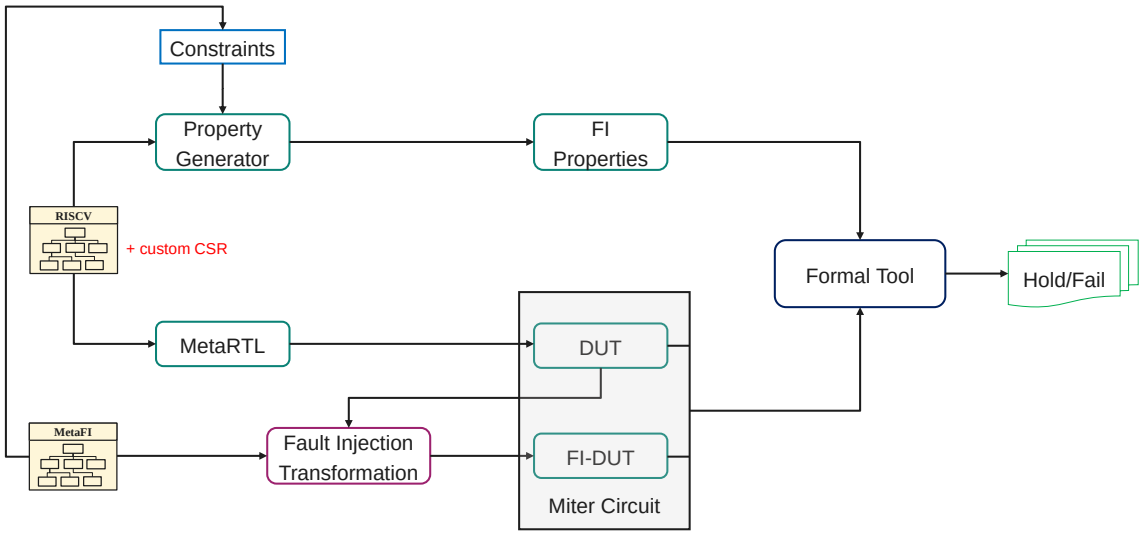


Figure 8.2: Setup of the DUT and properties

At the same time, the RISC-V metamodel serves as the specification for the property generation flow. A set of *functional constraints*, i.e., *assume* type properties, is generated for the RISC-V CPU complying with the model specifications. These kind of constraints prohibit the formal tool to consider illegal scenarios, e.g., illegal opcode, illegal operations etc.. Furthermore, another set of *test constraints* has been automatically added to the property generator to ensure an independent SBST. The constraints are summarized as following:

- External interrupts are disabled such that the control flow is not impacted.
- Control flow instructions such as branch or jump are disabled, thus SBST is independent of its memory location.
- The PC is constrained to increase linearly, i.e., PC+2 for compressed instructions and PC+4 for regular instructions.
- A single fault is injected per property.
- CSR instructions are disabled except custom CSR and PFC CSR. This is done such that the SBST does not change CSR states of the original program.

Contrary to specifying constraints via a hardware module as presented in [60], these constraints are expressed via *assume* properties. After specifying constraints, another property of *assert* type is created to check the fault effect on the miter circuit. This property pseudocode based on SVA format is shown in Figure 8.3. This property simply checks whether the designs are equivalent in presence of a single fault in the FI-DUT. The fault is injected via *fault_injection_macro*. This macro constraints the fault control signal to inject only a single stuck-at fault for the whole duration of the property, similarly to the macro presented in Algorithm 7. If the injected fault affects the FI-DUT, then the property would fail and the formal tool would generate a counterexample containing the instructions sequence. If the property holds, the injected fault is undetectable, thus there exists no pattern that can test the fault.

```
property generate_test_pattern;
@ (posedge clock) disable iff (reset)
    fault_injection_macro |-> DUT.Outputs == FI_DUT.Outputs;
endproperty
```

Figure 8.3: Fault Injection Property

8.2.2 Test Pattern Generation Flow

In general, it is very time-consuming to generate test patterns for several thousands of faults only by injecting faults via formal properties. In this thesis, fault simulation and formal properties are combined together to speed up the test pattern generation and to create a complete test program.

Figure 8.4 displays the overall flow of combining fault simulation and formal properties to generate the complete test program.

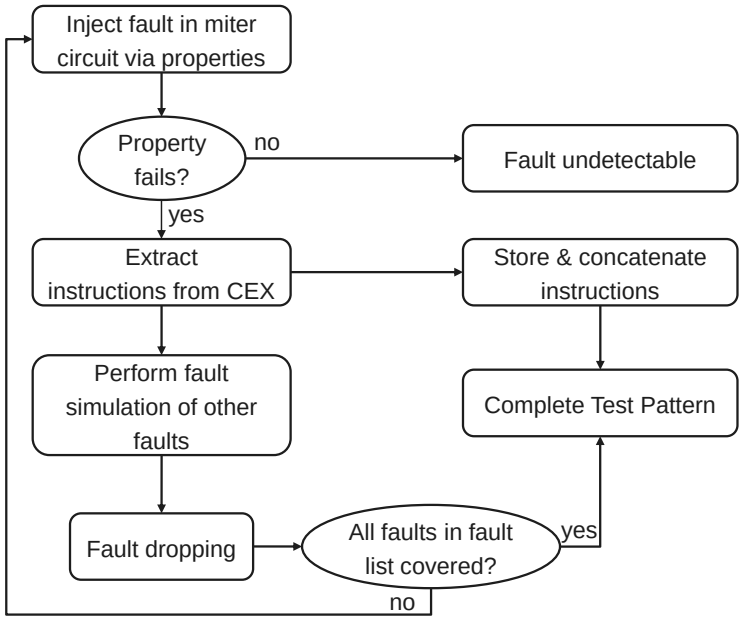


Figure 8.4: Complete Test Pattern Generation flow

Initially, the first fault in the fault list is injected via the macro in the property. A timeout mechanism is set in the formal tool, such that if the property does not fail within a time boundary, the fault is considered undetectable, i.e., the fault does not affect the design. Whenever the property fails, i.e., the fault affects the design, a counterexample is generated by the formal tool. The counterexample represents a input sequences, i.e., the test pattern, that can test the specific injected fault. This counterexample is represented via a text file and the sequence is stored in a hexadecimal format (.hex). At the same time, a testbench is generated by MetaFI framework to inject all faults in the design as explained in Chapter 4. A Tcl script extracts the input sequence from the counterexample file and stores it into another file named as *test_pattern_file.hex* and having the custom CSR as the first instruction. Moreover, this Tcl script modifies the generated

testbench and set the extracted input sequences as stimuli to the CPU inputs. The simulator is also automatically launched via another script and fault simulation is automatically performed by injecting all other faults using the same pattern as stimuli. Anytime another fault is tested by the same pattern, this fault is dropped and removed from the fault list. *Fault dropping* is a common term in the testing domain that refers to a practice where the fault is excluded from the test pattern generation. As a result, with the same test pattern, several faults can be detected. After fault simulation of all faults is complete, the next fault on the fault list is injected via formal properties. The next generated pattern from the counterexample is stored in the *test_pattern_file.hex* file after the previous pattern, i.e., patterns are concatenated. Fault simulation is performed again and the complete process is repeated until all faults have been tested. After the process is complete, the *test_pattern_file.hex* file contains the complete test pattern. An example how the complete test pattern looks like is represented visually via Figure 8.5. The normal CPU operation is interrupted and its content flushed via NOP instructions. After jumping to the test routine, a custom CSR that resets the Register File is fetched before each individual pattern such there are no dependencies between patterns.

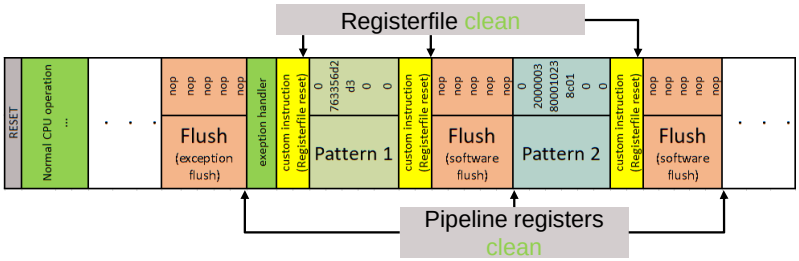


Figure 8.5: Pattern concatenation example

8.3 Program Flow Checking

Many faults in the CPU would cause anomalies in the program execution such as incorrect instruction order, modified instructions, illegal instructions, address misalignment of memory type instructions etc. Program Flow Checking (PFC) is a common safety mechanism to detect errors in the program flow [21]. The main idea behind PFC is to monitor the program execution by inserting signatures into the program during compile-time and checking them during run-time. The signatures are created via a hash (signature) function such as CRC. Generally, the flow checking process consists of: (i) creating the initial signature, (ii) updating the signature by hashing the current instruction and the previous signature according to the hash function, and (iii) verifying the signature. The verification is done by comparing the computed signature during the compile-time with the one computed during the run-time. If a hardware fault affects the program, the run-time signature would be different from the compile-time, thus a fault would be detected.

The run-time signature can be calculated via software but slows down the performance. Therefore, in this thesis a hardware PFC module is utilized that actively computes the run-time signature. The PFC hardware requirements are defined in the CPU metamodel and it is automatically added to the CPU. The hardware module is located in the Execute stage and hashes

the executed instructions. The state-of-the-art PFC techniques focus solely on the instructions signatures since the compile-time signature can be computed on the disassembly file of the compiled code. This thesis extends the existing PFC techniques by not only comparing the instruction signatures but also comparing signatures of ALU outputs or Register File outputs. Since the counterexamples are generated by the formal tool, not only test pattern instructions are accessible but also internal signals can be accessed, something that cannot be known on the dissasembly file. These internal signals, e.g., ALU outputs or Register File outputs, are stored into another file and the signature is computed over their values.

In the following a more detailed explanation of PFC hardware and how it is utilized to detect faults is given.

8.3.1 PFC hardware

The PFC hardware is automatically added to the processor core and it is located at the same pipeline stage as the respective signal whose signature is being calculated. The PFC module utilizes a CRC function to hash the data and its features, e.g., CRC polynomial. The function is defined in the RISC-V CPU generation framework.

Figure 8.6 displays the PFC hardware module. The PFC state register, i.e., the signature register that stores the run-time signature, is added to the RISC-V CSR interface. The extensability feature of RISC-V allows ISA extension, thus an extra CSR address in the custom read/write CSR is allocated for the PFC. The PFC is enabled when *pfc_start* is activated and the PFC state is initialized with the *initial_state* data. *pfc_start* control signal is high when the custom PFC CSR instruction is executed and *initial_state* represents the 32-bit value in the register in the Register File as defined by the PFC CSR instruction. Initialization immediately activates the data stream hashing, i.e., the next data after the PFC instruction is the first one entering the stream hash. This data is denoted by signal *data_in* and it represents the instruction being executed, ALU outputs or Register File outputs, i.e., it represents the signal that is selected to be hashed. The Hash Function, i.e., hardware-based CRC implementation, calculates the next signature (*signature_out*) based on the *data_in* and the current value of the PFC State register. Data stream checking is deactivated (PFC-off domain) by using a selected register as source in the PFC instruction. Deactivation is represented by the signal *pfc_end*. This instruction is the last data that enters the hash signature. The PFC State value is static from then on and can be read.

8.3.2 Fault detection flow

The PFC hardware flow provides a fast run-time signature calculation thus enabling an efficient fault detection. The main idea behind the fault detection flow is to compare a precomputed signature with the one computed during run-time by the hardware. By doing so, any mismatch between the signatures signals a detected hardware fault. Figure 8.7 illustrates the steps of the fault detection flow using PFC. The *PFC Start* which is represented via the custom PFC CSR instruction activates the PFC hardware module to start hashing of the upcoming data. The upcoming data that will be hashed is represented by the generated test pattern instructions (see above for the test pattern generation flow). The test pattern instructions are stored in a predetermined memory location and they are immediately fetched by the CPU after the custom CSR

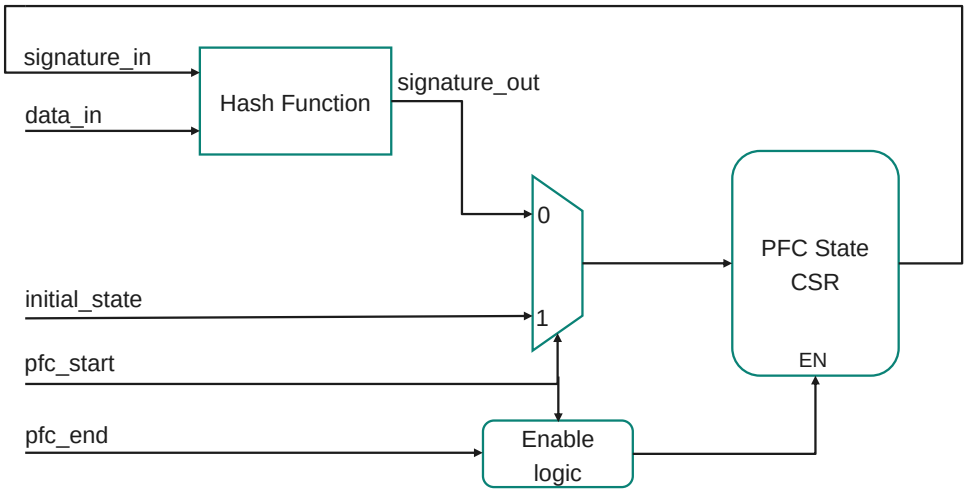


Figure 8.6: PFC hardware module

instruction. After all test instructions have been executed, another custom PFC CSR instruction (*PFC End* in the Figure 8.7) signals the PFC hardware module to stop hashing. This instruction is the last one that enters the hash signature. The next instruction is not hashed any more and the state of the signature is static from then on and it is stored in the PFC CSR register. This way, the signature value can be read using a CSR read instruction.

Meanwhile, the static expected signature value is precomputed via software methods. A python script performs hashing statically over the generated test pattern using the same CRC polynomial as the hardware PFC module. Normally, the hardware module should produce the identical signature if there is no fault on the design. If a fault occurs, the run-time signature is different from the static one, resulting in a detected fault. After hashing is finished, a comparison is made between the signatures. The run-time signature is read via a PFC CSR read-instruction. Finally a comparison instruction compares this value with the static precomputed signature. Therefore, after the PFC stop instruction, two more instructions are added to the test pattern for comparison reasons.

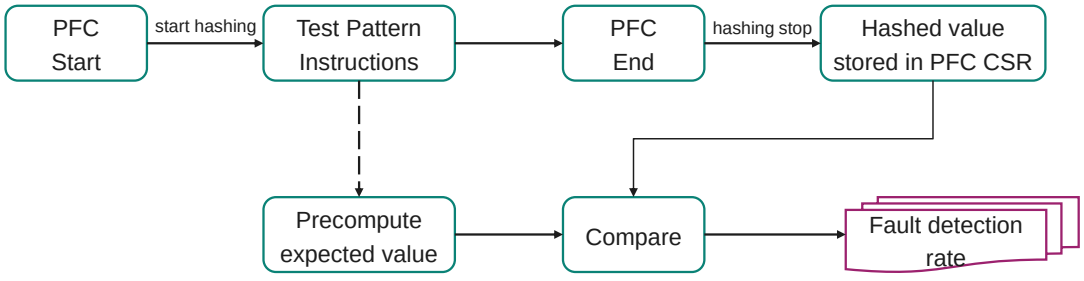


Figure 8.7: Fault detection via PFC

Chapter 9

Experimental Results and Discussions

This chapter presents various hardware architectures that are used as testing ground for the techniques described previously in this thesis. Initially, the performance of the mixed granularity RTL models is evaluated, and then this technique is used to perform safety analysis of two different designs. Afterwards, the fault emulator is thouroughly analyzed and data regarding its area, speedup and performance are analyzed. Next, different design architectures undergo various fault injection campaigns and the results are interpreted and analyzed to provide the design fault propagation rate for these designs. The chapter concludes with results and applications regarding formal-based fault propagation analysis and SBST generation.

9.1 Fine-grained RTL Models Performance

9.1.1 Experimental Setup

To measure the performance of the mixed granularity models approach, a CPU subsystem was generated using MetaRTL, as illustrated in Figure 9.1.

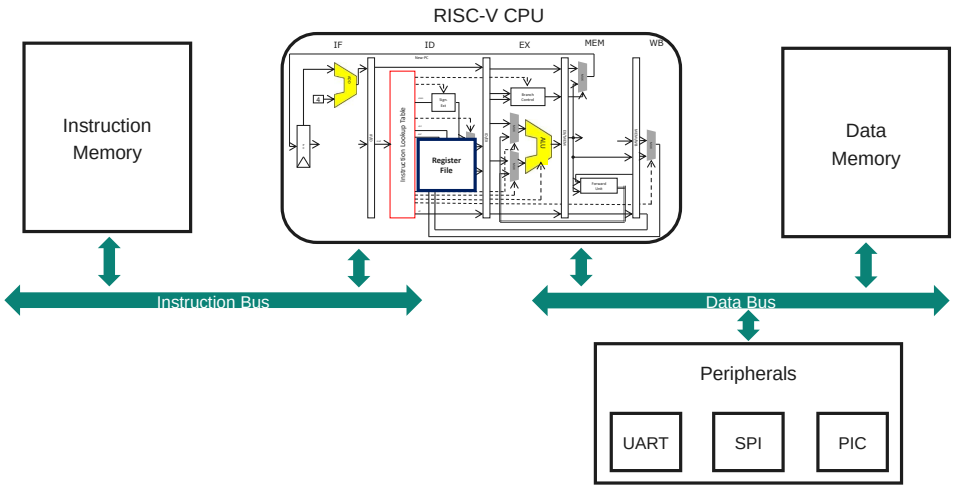


Figure 9.1: RISC-V based CPU subsystem

The CPU subsystem is composed of a RISC-V based CPU that adheres to a Harvard archi-

tecture, separating its Instruction Memory and Data Memory domains. The five-stage pipelined CPU offers the versatility of supporting both synchronous and asynchronous exceptions. Moreover, the CPU is equipped with different safety mechanisms such as CRC to protect against various faults. Additionally, the subsystem includes an assembly of peripherals including UART, SPI, and a Programmable Interrupt Controller (PIC).

The firmware is loaded into hte instruction memory. The firmware program is automatically generated via the underlying firmware generation framework. During the experiments, two different firmware programs were utilized such as: (i) Program Flow Monitoring and (ii) Firmware Verification. The program flow monitoring can identify potential control faults arising from incorrect program flow execution. This entails checking the order in which instructions are executed. In a similar fashion, the Firmware Verification program detects possible program execution faults. The technique differs from Program Flow Monitoring because the application designer takes charge of initializing variables and specifying the expected values for the variables that require verification.

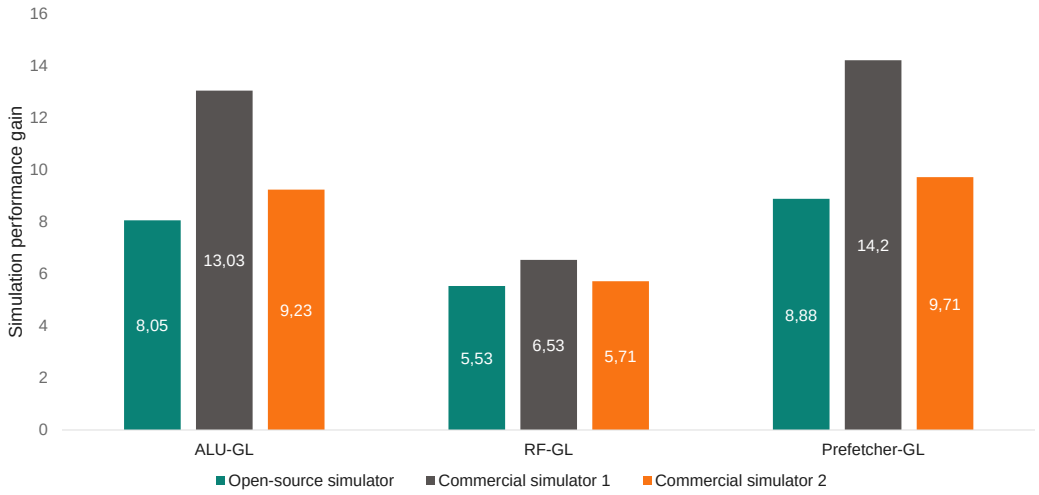
9.1.2 Performance Evaluation

The design is simulated for 10k clock cycles and the Firmware Verification program is employed. In the simulation, the CPU core is initially implemented at the gate level granularity, while other aspects of the SoC, like peripherals and memories, are represented using traditional RTL to reduce complexity. As discussed in Chapter 5, fault simulation campaigns may not require complete gate level design simulation, therefore mixed-granularity versions are generated as well. The fault simulation testbenches are automatically generated utilizing the fault handling framework. Since the techniques discussed in this thesis are not tied to a specific simulator, experiments were run on three different open-source and commercial simulation tools

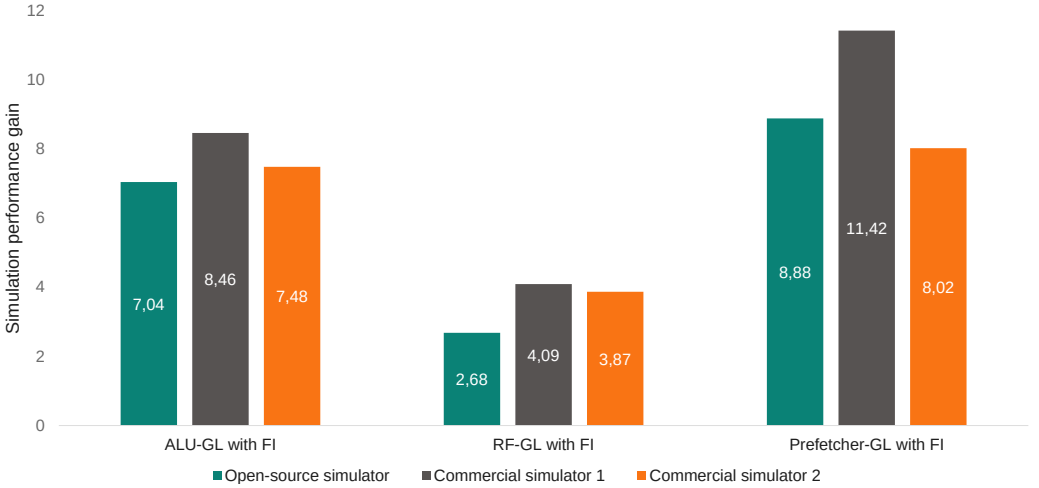
To evaluate the overall simulation performance improvement, the SoC is re-simulated, this time, instead of a full gate level simulation, only particular CPU components at the gate level granularity are simulated such as: (i) ALU, (ii) Register File, and (iii) Prefetcher. Table 9.1 displays the runtimes of running a simulation of designs for 10k cycles. The simulation was performed on: (i) a full gate level description of the CPU, (ii) a mixed granularity CPU with only the respective component described in gate level, and (iii) a mixed granularity CPU with the respective component in gate level equipped with fault injectors. Three different simulators were utilized, demonstrating the genericity of the approach. The runtime depicted in the table aggregates elaboration, compilation, and simulation times. As can be seen in the table, a notable performance gain is achieved through simulation of mixed-granularity RTL models. This performance gain is depicted also via the charts in Figure 9.2. Figure 9.2a displays the gain when performing simulation on mixed granularity models without fault injectors, while Figure 9.2b illustrates the results when fault injectors are inserted into the design. As can be observed from the charts in Figure 9.2a, the highest performance gain is achieved when utilizing commercial simulator 1 with a gain factor of 14.2. The lowest performance gain is achieved while using open-source simulator with a minimal gain factor of 5.53. In Figure 9.2b, it is observed that the overall gain is lower after adding saboteurs due to the increased design area. Nevertheless, there remains a performance gain compared to a full gate level simulation. A minimal performance gain of 2.68 and a maximum gain of 11.42 was achieved.

Table 9.1: Simulation runtime of mixed granularity models

GL component	Simulation tool	Runtime (h:m:s)		
		Full GL CPU	Mixed CPU	Mixed-CPU with FI
ALU	Open-source tool	1:43:20	12:50	14:40
	Commercial tool 1	1:20:00	6:08	9:27
	Commercial tool 2	9:14	1:00	1:14
RF	Open-source tool	1:43:20	18:41	38:30
	Commercial tool 1	1:20:00	12:15	19:33
	Commercial tool 2	9:14	1:37	2:23
Prefetcher	Open-source tool	1:43:20	11:38	16:50
	Commercial tool 1	1:20:00	5:38	7:00
	Commercial tool 2	9:14	0:57	1:09



(a) gate level model of component without fault injectors



(b) gate level model of component with fault injectors

Figure 9.2: Simulation performance gain of mixed RTL models

9.1.3 Application

Fault simulation on mixed granularity models was utilized to evaluate software-based safety detection mechanisms such as: (i) Program Flow Monitoring and (ii) Firmware Verification. The utilized CPU subsystem is identical to the one displayed in Figure 9.1. Since both of these programs monitor and detect faults of the instruction sequences, the *Prefetcher* block was selected as subject of fault injection due to its ability to control the PC register. In the Flow Monitoring Program, a fingerprint verification module triggers a fault signal whenever the PC value deviates from the expected value. This fault signal is closely monitored during simulation to detect any faults by selecting it as a checker strobe in the Fault Handling metamodel. For the DUT equipped with firmware verification, fault detection involves comparing a calculated Fibonacci number and the received byte with their expected values.

A total of 1000 random single faults was injected at various time points into both the DUTs; once on a full CPU on a gate level granularity and once only the *Prefetcher* on a gate level granularity. The results of the fault simulation campaign are presented in Tables 9.2 and 9.3.

Table 9.2: Fault simulation of CPU-subsystem running program flow monitoring

Component	Runtime h:m:s	Clock cycles	Injected faults	Failures	Detected failures
GL Prefetcher	1:15:01	9k	1k	675	221
GL CPU	4:40:04	9k	1k	675	221

Table 9.3: Fault simulation of CPU-subsystem running firmware verification

Component	Runtime h:m:s	Clock cycles	Injected faults	Failures	Detected failures
GL Prefetcher	6:48:04	65k	1k	574	477
GL CPU	14:50:50	65k	1k	574	477

As can be seen from the tables, the runtime of a full gate level simulation is larger compared to a mixed granularity simulation; up to 2.1-3.5 times slower. The Flow Monitoring DUT was simulated for 9k clock cycles (CC) and 675 faults propagated as failures. Out of these failures, 221 were detected by the program. Meanwhile, the firmware verification DUT was simulated for 65k clock cycles (CC) and 447 failures were detected out of a total 574 failures. Therefore, Firmware Verification program was able to identify more faults compared to the Flow Monitoring Program, but this came at the cost of increased runtime, measured in hours. It is evident that an equal number of faults are both propagated and detected for both mixed granularity simulation and full gate level simulation, demonstrating that employing mixed granularity fault simulation does not mask the faults.

9.1.4 Discussions and Observations

The techniques discussed in this thesis are not tied to a specific simulator, making them universally applicable to fault simulation campaigns utilizing any RTL simulator. The presented

results demonstrate a notable reduction in effort when using mixed granularity compared to full gate level granularity simulation. However, it is worth noting that the saboteur-based fault injection technique introduces a certain level of overhead due to the increased execution of boolean operators. Nevertheless, the effort required even with fault injectors remains significantly lower compared to full gate level simulation without any fault injection mechanisms.

9.2 Analysis and Performance of Fault Emulator

9.2.1 Experimental Setup

The proposed fault emulation framework's scalability, applicability, and effectiveness were demonstrated through experimentation on a CPU subsystem, referred to as DUT, which consists of a single 5-stage pipelined RISC-V CPU (IF, ID, EX, MEM, WB), as well as Instruction Memory, Data Memory, and various peripherals including buses, UART and PIC. Fault injection was carried out on a randomly selected sequence of instructions. For experimentation, an FPGA target implementation was utilized, specifically the Digilent™ VC707 Board in conjunction with Vivado 2020.1. The emulation ran at a fixed frequency of 20 MHz, and simulation experiments for comparison were conducted using the Xcelium® simulator. The primary development environment employed is a 64-bit version of Red Hat Enterprise Linux™ (RHEL) 7 and the computing infrastructure is integrated into a Load Sharing Facility (LSF) Compute Farm. The utilized application firmware is a Pulse Width Modulation (PWM) program.

9.2.2 Hardware Utilization

Table 9.4 displays the FPGA resource utilization metrics for the DUT in its original configuration, where no fault saboteurs have been introduced, and without hardware modifications to perform fault emulation. The BRAM Primitives in use contain the compiled firmware.

Table 9.4: Original DUT resource utilization

Slice LUTs	Slice Registers	BRAM primitives
8802	6791	14

The insertion of fault injectors into the initial DUT results in additional area overhead, and the inclusion of auxiliary hardware modules to facilitate fault emulation further contributes to the overall area increase. Table 9.5 shows the resource utilization resulting from the insertion of fault injectors into different components of the processor. As the fault injectors are essentially combinatorial structures their introduction into a design primarily impacts the utilization of Look-Up Tables (LUTs). Some slight fluctuations are also noticeable in the Slice Register values, which can be attributed to internal optimizations conducted by the Vivado tool during synthesis. The tabular data clearly illustrates a linear correlation between the growth of fault control lines and the corresponding utilization of LUTs. This correlation is visually depicted in Figure 9.3. In summary, the resource utilization exhibits a primarily linear pattern as the design size, with injected faults, expands, and it does not experience disproportionate expansion.

Table 9.5: Resource utilization of DUT with fault injectors

Selected component for fault injection	Fault control lines	Slice LUTs	Slice Registers	BRAM Primitives
Hazard Detection Unit	63	9061	6791	14
Program Counter	109	9223	6790	14
WB stage	262	9615	6791	14
ALU result	613	10291	6796	14
Forwarding Unit	786	10519	6795	14
MEM stage	954	10708	6794	14
Program Flow Check	1051	11130	6796	14
Instruction Decoder	1865	12485	6790	14
Event Counters	1919	12667	6795	14
Prefetcher	2125	13033	6790	14
ALU	2708	14672	6796	14
Exception Unit	5008	18766	6766	14
Control & Status Registers	6371	21791	6795	14
Register File	7679	24389	6793	14
IF stage	7952	24389	6793	14
ID stage	10948	29867	6792	14
EX stage	15826	43044	6793	14

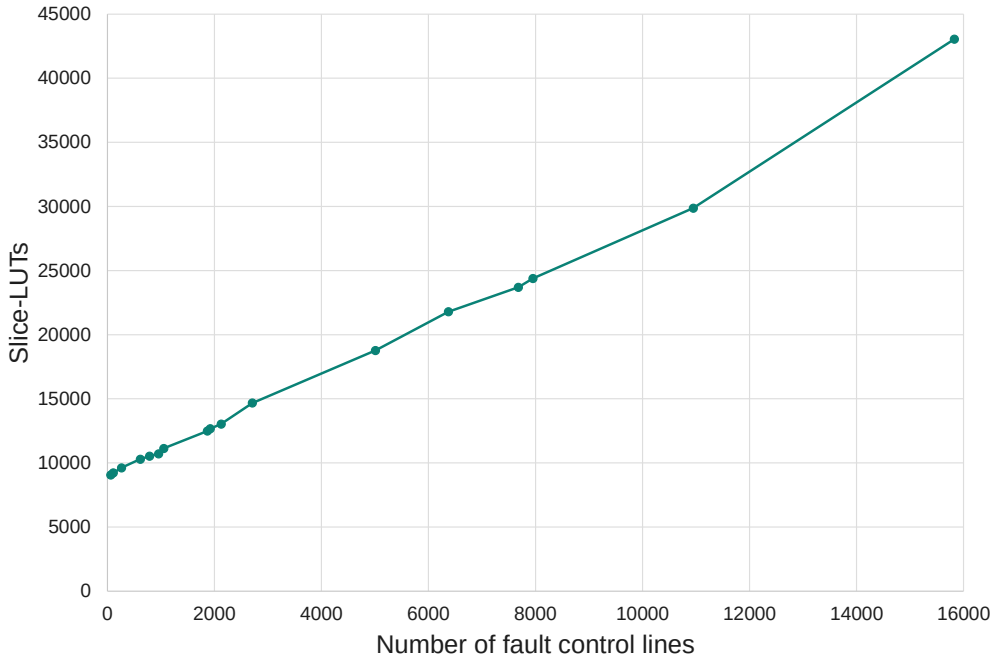


Figure 9.3: Slice-LUTs utilization of DUT with varying fault control lines

As illustrated in Chapter 6, the execution of fault emulation requires the addition of essential auxiliary hardware modules, namely the Fault Controller, Postprocessing block, and Data harvesting logic. The integration of these auxiliary modules introduces an additional layer of

resource utilization overhead. To provide a precise information of this overhead, fault emulation campaigns were conducted on each individual stage of the processor. The emulation duration was set at 20k clock cycles, and both stuck-at-0 and stuck-at-1 fault models were injected. Table 9.6 displays the resource utilization for the aforementioned campaigns. As can be seen, there is a consistent rise in the count of Slice LUTs as the number of fault control lines increases. Furthermore, the increment in Slice Registers can be attributed to the Data Harvesting Logic, which accommodates a greater volume of data as the number of injected faults increases. Nevertheless, the increase of Slice LUTs and Slice Registers resources is in accordance with the scale of fault-injected designs, primarily adhering to a linear progression as depicted in Figure 9.3. The auxilliary hardware modules add a maximum overhead of 53% in terms of Slice-LUTs and a maximum overhead of 65.2% in terms of Slice Registers. Notably, the count of BRAM Primitives remains constant with an overhead of 57.1%, as the emulation duration remains uniform across all experimented modules. This uniformity ensures that the traces generated are of the same length.

Table 9.6: Resource utilization of the fault emulator for various fault emulation campaigns

Selected component for fault injection	Fault control lines	Number of injected faults	Slice LUTs	Slice Registers	BRAM Primitives
WB stage	262	493	10218	7883	22
MEM stage	954	1298	12508	9029	22
IF stage	7952	12799	37316	9179	22
ID stage	10948	16803	44387	9205	22
EX stage	15826	27616	62519	11222	22

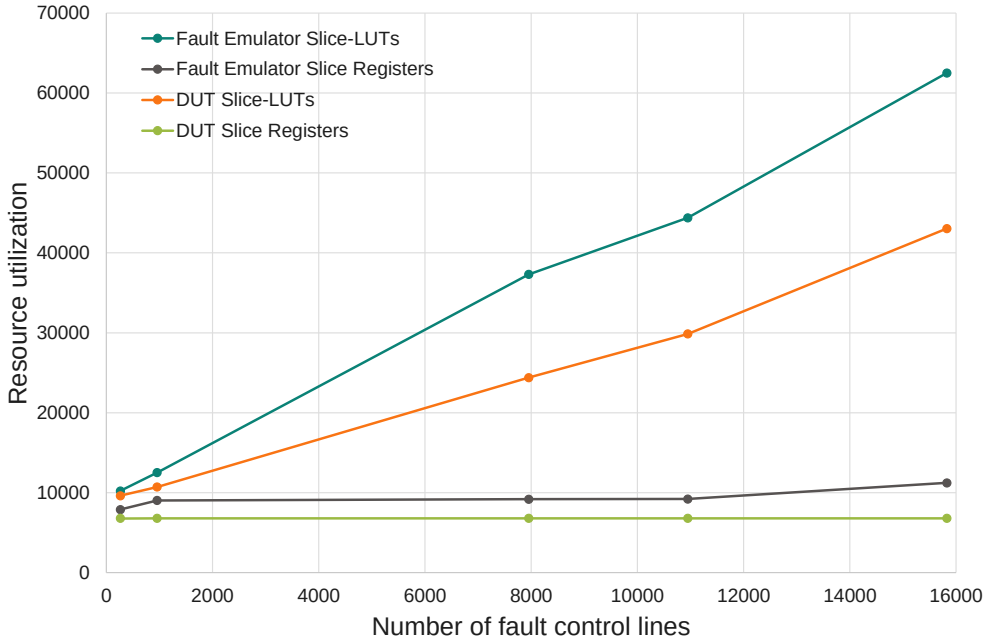


Figure 9.4: Resource utilization of fault emulator for varying fault emulation campaigns

9.2.3 Performance Evaluation of the Fault Emulator

The central focus of the developed fault emulator is to reduce the runtime of fault injection campaigns through the adoption of emulation-based methodologies. In this context, experiments were conducted, encompassing both fault emulation and simulation workflows, thereby enabling a comparative analysis. The experimental setup is identical to the one presented in the previous subsection; experiment's duration is 20k clock cycles and faults are injected into individual processor stages. Table 9.7 provides comprehensive data of experimental outcomes including both emulation and simulation techniques. As can be seen from the table, the number of injected faults exhibits a sharp increase, which consequently leads to a notable escalation in simulation runtime. In contrast, emulation runtime demonstrates a more consistent growth as the design size expands and the number of injected faults increases. Notably, for smaller fault-injected modules like the WB Stage, and to some extent, the MEM Stage, the emulation runtime is not much different to its simulation counterpart, as illustrated graphically in Figure 9.5.

Table 9.7: Runtime values for fault emulation and simulation experiments

Selected component for fault injection	Fault control lines	Number of injected faults	Emulation runtime h:m:s	Simulation runtime h:m:s	Performance gain
WB stage	262	493	00:25:38	01:17:05	3.10
MEM stage	954	1298	00:35:38	02:25:43	4.15
IF stage	7952	12799	01:47:44	12:13:53	6.86
ID stage	10948	16803	02:09:30	29:25:17	9.00
EX stage	15826	27616	03:09:56	23:30:53	7.46

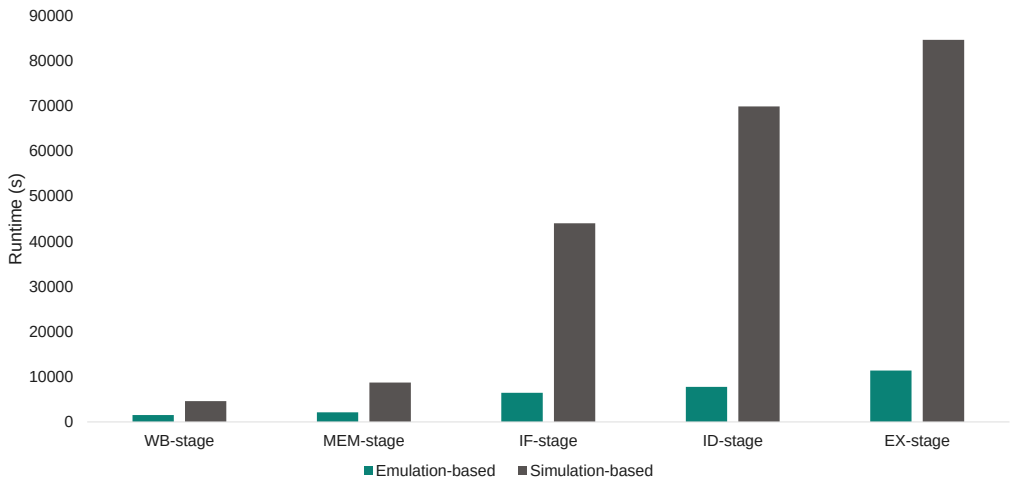


Figure 9.5: Runtime comparisons of emulation and simulation experiments

This similarity can be attributed to the conventional design approach, where the last stage (WB) of a pipeline is purely combinatorial. This design prevents operations from extending into

subsequent clock cycles, and as a result, simulation tools exhibit a good performance, resulting in reduced runtime.

Another substantial portion of the emulation runtime can be attributed to the bitstream generation time. This component is predominantly governed by the internal processes of the Vivado toolchain, thus cannot be optimized or controlled due to its IP-protected nature. The difference between bitstream generation and the emulation itself can be as substantial as 300-fold.

9.2.4 Performance Evaluation of the "On-the-Fly" Emulation Technique

Chapter 6 presents a novel fault emulation technique where the comparison of the faulty and non-faulty emulations is done in parallel, i.e., "on-the-fly". This technique reduces BRAM utilization, thus accomodating experiments involving more extensive instruction traces. Leveraging this capability, the emulation time for the similar experimental setup has been extended to 100k clock cycles. Table 9.8 provides runtime data for fault emulation experiments conducted with the "on-the-fly" technique. As observed in the table, the runtime values of emulation experiment exhibit minimal changes when compared to the values provided in Table 9.7, even with significantly extended trace lengths. Conversely, simulation values experience a larger increase, particularly for larger modules such as the EX Stage. Therefore, optimal performance gain can be attained by conducting experiments with larger designs, longer instruction traces, and a greater number of injected faults.

Table 9.8: Runtime values for fault emulation and simulation experiments using "on-the-fly" technique

Selected component for fault injection	Fault control lines	Number of injected faults	Emulation runtime h:m:s	Simulation runtime h:m:s	Performance gain
WB stage	262	493	00:27:14	03:51:16	8.56
MEM stage	954	1298	00:38:10	09:23:56	14.82
IF stage	7952	12799	02:27:50	42:20:56	17.28
ID stage	10948	16803	03:13:50	67:17:54	20.91
EX stage	15826	27616	03:33:57	168:52:18	47.57

9.2.5 Emulation-based Fault Propagation Analysis

Following fault emulation of the components presented in Table 9.7, the data transmitted from the FPGA to the Host-PC undergoes processing to generate fault classification information. Figure provides a detailed breakdown of fault classifications for faults injected into the pipeline stages using EFI emulation. These stages encompass various arithmetic and logical components that impact the POs and Register File outputs. Figure 9.6 displays the fault propagation analysis of faults injected into different pipeline stages.

A significant observation across all three cases is the predominant classification as *Safe* faults. This phenomenon can be attributed to the firmware employed in the emulation, which serves as an application firmware with limited capacity to sensitize specific paths in the design. Consequently, the fault effects do not propagate to the POs. However, certain stages contain

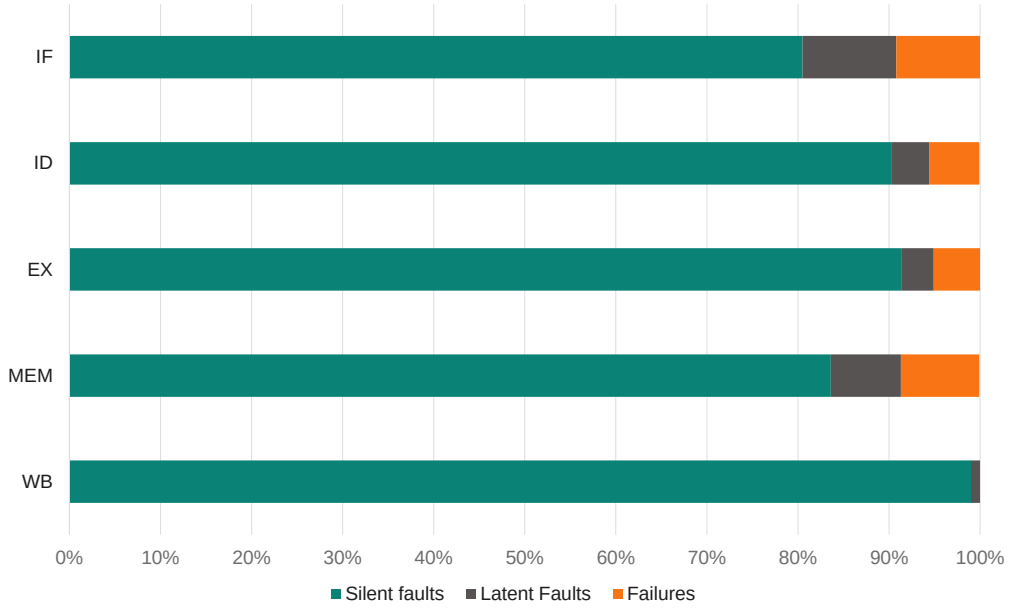


Figure 9.6: Emulation-based fault propagation analysis of different pipeline stages

modules crucial to the execution flow, either directly or indirectly. As a result, faults injected into these vital modules tend to manifest their effects on the POs. Notably, the PC computation logic in the IG Stage demonstrates a higher percentage of *Failures* and *Latent* faults compared to the other stages. Additionally, the ID Stage contains the Register File, which receives numerous computation outputs during execution, leading to a higher proportion of *Latent* faults compared to the EX Stage. The increased prevalence of *Failures* and *Latent* faults within the MEM Stage can be rationalized by its relatively smaller size, resulting in a reduced number of faults within its verification scope. In contrast, the WB Stage, characterized by its purely combinatorial nature, consistently prevents the propagation of fault effects, resulting in a predominant classification as *Silent* faults.

9.2.6 Discussions and Observations

The architectural framework introduces a manageable linear overhead to the overall hardware resource utilization, showcasing its ability to scale effectively when applied to a complete CPU subsystem. Furthermore, the emulation framework delivered noteworthy performance enhancements when operating at significantly higher frequencies than conventional commercial simulators. It is worth emphasizing that the generation of the bitstream constituted the most time-consuming phase of the emulation campaign, whereas the actual fault emulation process took only a matter of seconds. Moreover, it is essential to highlight that the performance gains exhibited a positive correlation with longer emulation durations. Consequently, the proposed emulation-based approach proves to be particularly well-suited for executing larger instruction sequences.

9.3 Case study: Statistical-based Fault Propagation Analysis

9.3.1 Experimental Setup

Two distinct RISC-V-based CPU subsystems were created using MetaRTL. A 2-stage pipelined CPU and a 5-stage pipelined CPU were included as DUTs, along with components such as RV32-IMC CPU, Instruction Memory, Data Memory, Instruction and Data Bus, and Peripherals. The specification of the CPUs was defined using the RISC-V metamodel (see Figure 7.5), which allowed for the generation of different RISC-V variants by adjusting parameters like supported instructions and extensions. Specific components like the ALU, Instruction Decoder, and Branch Control Unit were targeted for fault injection experiments to achieve SFI. These components were described at a gate level granularity, while the rest of the DUT remained at the original RTL granularity. Random fault models were injected at random locations during random simulation times, as SFI is applicable to random faults. The quantity of faults to inject was automatically determined by the framework based on the desired level of confidence and error margin, as shown in Table 4.1. The only inputs required from the user were the simulation duration, level of confidence, and error margin.

9.3.2 CPU Workloads for Fault Propagation Analysis

To analyze fault propagation, four different benchmarks were carefully chosen to simulate realistic fault scenarios. These benchmarks cover a variety of computational domains and provide a comprehensive assessment of the system's behavior under different conditions. Each benchmark was specifically selected for its relevance and specific characteristics. The first benchmark used is the well-known Dhrystone benchmark, initially created by Reinhold P. Weicker in 1984. Dhrystone is often used as a synthetic benchmark to evaluate the performance of integer programming in a system. Although it is relatively simple by today's standards, its enduring popularity and the high percentage of control flow operations it involves make it an exciting workload for fault propagation analysis. Dhrystone is included in this study due to its ongoing importance. The other three benchmarks incorporated into the study are cryptographic algorithms serving unique roles in fault propagation analysis. These benchmarks - SHA-256, MD5, and CRC-32 - represent cryptographic hash functions and checksum algorithms. They were selected from the Embench suite, a collection of C benchmarks consisting of 22 real-world applications commonly used in deeply embedded systems. These cryptographic algorithms were chosen due to their widespread use and the significant security implications associated with faulty behaviors. As these algorithms play a critical role in ensuring data integrity and confidentiality, their inclusion in the analysis is essential. These algorithms present unique challenges in terms of fault tolerance and error propagation, making them valuable additions to the suite of benchmarks for fault analysis.

All the benchmarks in the study were compiled using GCC 11.1 at an optimization level of -O3. Additionally, the newlib 4.1.0 library was configured with an optimization level of -O2 and garbage collection.

(a) Dhrystone instruction statistics

(b) SHA-256 instruction statistics

(c) MD5 instruction statistics

(d) CRC-32 instruction statistics

Figure 9.7: Instruction statistics for different benchmarks

9.3.3 Fault Propagation Analysis

Figures 9.8-9.11 depict the fault propagation rate, which refers to the proportion of faults that propagate to the primary outputs of the CPU, during the execution of SFI on the three mentioned components. The SFI process was carried out with a 5% error margin and three varying confidence rates: 95% (with 384 injected faults), 99% (with 663 injected faults), and 99.8% (with 955 injected faults). The simulation was conducted over a total of 37868 clock cycles.

Table 9.9 shows the total number of fault locations for each of the three components, namely the ALU, Decoder, and Branch Control Unit. As observed from the table, the ALU of the 5-stage CPU exhibits 8.1% more fault locations compared to the 2-stage CPU, while there is only a slight increase in the number of locations for the 5-stage CPU Decoder. Conversely, there is no change in the number of locations for the Branch Control Unit.

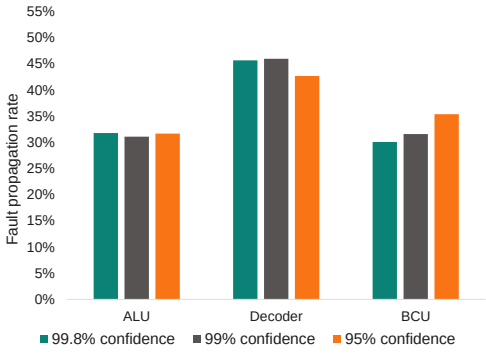
Table 9.9: Total number of fault locations

Component	2-stage pipelined CPU	5-stage pipelined CPU
ALU	5376	5812
Decoder	3504	3514
Branch Control Unit	754	754

The results from Figures 9.8-9.11 demonstrate that even though the confidence level of SFI campaigns for each component varies, the fault propagation rates consistently stay within the 5% acceptable error margin. This confirms the efficiency of the method employed. Notably, there is a slightly higher fault propagation rate (ranging from 5-10%) observed in the 2-stage

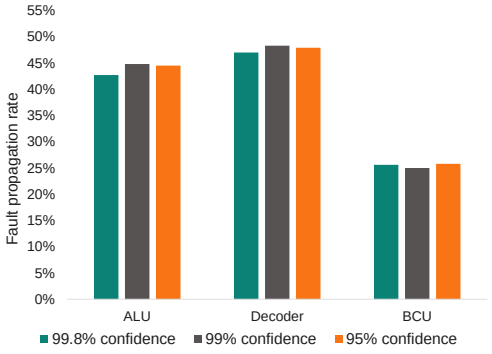
pipelined CPU compared to other benchmarks. This difference is due to the design's smaller area, which lacks additional logic to prevent numerous faults like the 5-stage pipeline CPU. Among all components, the 2-stage pipelined CPU running the Dhrystone benchmark has the highest failure rate, while the 5-stage pipelined CPU running the MD5 benchmark has the lowest failure rate. The decoder component is the most susceptible to failures, requiring protective measures. On the other hand, the branch control unit is the least vulnerable to faults since it relies on specific instructions like branch and jump types to activate faults. Nevertheless, the results show that the fault propagation rate depends on the workload, indicating its dependence on the firmware executed by the CPU.

(a) Fault injection on 2-stage pipelined CPU

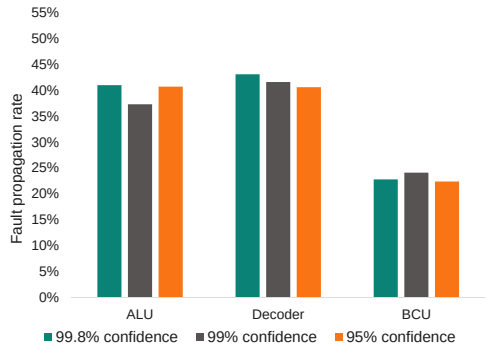


(b) Fault injection on 5-stage pipelined CPU

Figure 9.8: fault propagation rates using Dhrystone benchmark as firmware



(a) Fault injection on 2-stage pipelined CPU



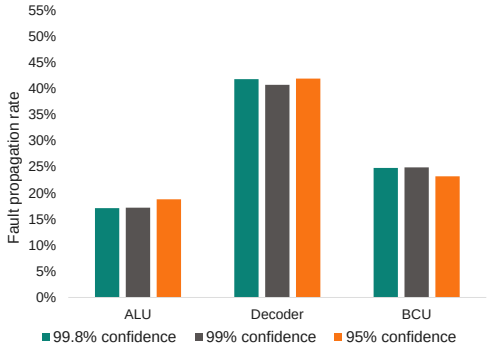
(b) Fault injection on 5-stage pipelined CPU

Figure 9.9: fault propagation rates using SHA-256 benchmark as firmware

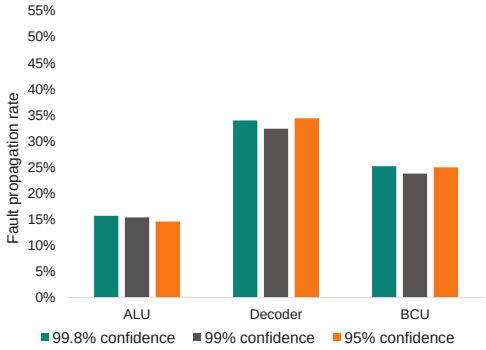
9.4 Analysis of Processor Safety Verification

9.4.1 Experimental Setup

The scalability, applicability, and efficacy of the processor safety verification (outlined in Chapter 7) have been demonstrated through experimentation on two distinct variants of 5-stage

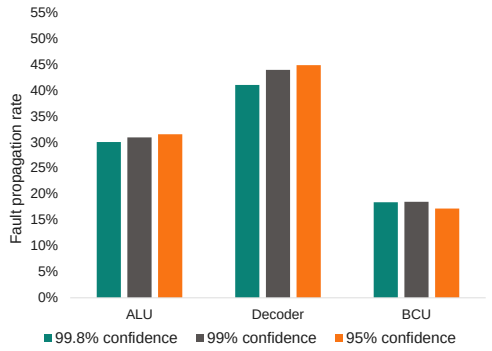


(a) Fault injection on 2-stage pipelined CPU

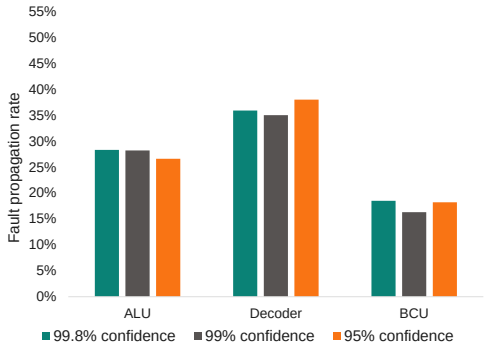


(b) Fault injection on 5-stage pipelined CPU

Figure 9.10: fault propagation rates using MD5 benchmark as firmware



(a) Fault injection on 2-stage pipelined CPU



(b) Fault injection on 5-stage pipelined CPU

Figure 9.11: fault propagation rates using CRC-32 benchmark as firmware

pipelined RISC-V processor cores: RV32IMC and RV32IMCZicsr. These processors are generated utilizing MetaRTL and incorporate key features such as a Harvard architecture, a 32-bit ALU, an in-order fetch unit, and a configurable multiply unit. It's worth noting that one of these variants had previously been implemented in an industrial System-on-Chip (SoC). As our formal tool, OneSpin 360 DV-Verify® by SiemensEDA was employed, operating on an 11th Gen Intel® Core™ i7-1185G7 @3.0GHz processor with 32 GB of RAM.

9.4.2 Processor Hardening Verification

Chapter 3 describes various techniques for hardening designs to mitigate the impact of faults. For the experiments, configurable TMR and SEC/SED ECC techniques were utilized due to their capacity to detect and correct faults. The workflow illustrated in Figure 7.6 was followed to harden the design and introduce fault injection capabilities. To verify the described safety mechanisms, FI-S2QED was employed for a hardened RISC-V (RV32IMC) processor core. Multiple components within the CPU, including the Register File, ALU, and pipeline registers, underwent hardening. Two experiments were conducted on the hardened processor: one with ECC and another with TMR applied. Each experiment was validated with a single property,

which was limited to inject a single fault model (the formal tool was capable of injecting the fault at all feasible design locations). The outcomes of these experiments have been presented in Table 9.10.

As evident from the table, two bugs were identified in the hardened CPU utilizing ECC. Upon thorough examination and debugging, it was determined that these anomalies were attributed to incorrect interconnections of clock and reset signals within sequential components, resulting in integration-related issues. Notably, the runtime for property failures was brief, lasting less than 30 seconds. In contrast, the employment of TMR revealed no bugs, and all injected faults were successfully corrected.

Following the successful fixing of the bugs, all injected faults were effectively corrected by the inherent mechanisms. Subsequently, a deliberate process of manually introducing faults into both TMR and ECC designs was undertaken. Notably, none of the safety mechanisms proved capable of correcting these manually injected faults, leading to the failure of FI-S^{2}QED in both scenarios. It's noteworthy that the manual effort needed during this process was minimal, with the majority of the time allocated to configuring model data and addressing the debugging outcomes.

Table 9.10: CPU hardening verification results

Mechanism	#properties	#faults injected	#bugs	Fail time
ECC	1	1	2	<30s
TMR	1	1	-	-
ECC + bugs inserted	1	1	1	<30s
TMR + bugs inserted	1	1	1	<30s

9.4.3 Processor Fault Propagation Analysis

In Chapter 7, the utilization of formal verification for the exhaustive analysis of fault propagation using k-FI-S^{2}QED was discussed. In the application of this technique, an in-depth fault propagation analysis was performed on eight distinct components, encompassing the ALU, Hazard Detection Unit (HDU), Decoder, Forwarding unit, Register File, and pipeline registers. These analyses were carried out on two variations of RISC-V CPUs such as RV32IMC and RV32IMCZicsr. Moreover, for the purpose of establishing a foundation for comparison, fault analysis was also conducted using simulation- and ATPG-based techniques.

Formal-Based Analysis

Table 9.11 and Table 9.12 provide an illustration of the fault coverage related to processor components when applying the k-FI-S^{2}QED method. In the tables, *DF* denotes detected faults, *RF* indicates redundant faults, *CEX length [min, max]* signifies the minimum and maximum number of instructions required for fault propagation (e.g., some properties require only 1 instruction to detect the fault, while others may require up to 5), and *Fail time* presents the shortest

and longest runtimes of individual properties for fault detection (excluding holding properties). The tables reveal that faults injected into the ALU, HDU, Register File, ID-EX, EX-MEM, and MEM-WB registers propagate to both CPU states and primary outputs for both the RV32IMC and RV32IMCZicsr variants. The fault coverage of the Forwarding Unit remains consistent in both variants, due to design redundancy that remains independent of the Zicsr extension. However, the number of faults in the Decoder differs between the two variants due to the additional decoding logic introduced by the Zicsr extension. Notably, an increase in fault coverage is observed for the ID-EX register due to faults injected into the CSR address lines, which subsequently determine the correct value of the CSR register to be written to the register file a few cycles later.

Table 9.11: RV32IMC fault propagation analysis

Component	# total faults	DF (%)	RF (%)	CEX length [min, max] instructions	Fail time (s) [min,max]
ALU	2689	100	0	[1,5]	[20,120]
HDU	63	100	0	[1,5]	[20,90]
Decoder	1681	77.98	22.02	[1,5]	[23,70]
Forwarding unit	786	91.86	8.14	[1,5]	[24,92]
Register File	1514	100	0	[1,5]	[20,120]
ID-EX register	694	93.5	6.5	[1,5]	[24,260]
EX-MEM register	199	100	0	[1,5]	[29,62]
MEM-WB register	308	100	0	[1,5]	[23,42]

Table 9.12: RV32IMCZicsr fault propagation analysis

Component	# total faults	DF (%)	RF (%)	CEX length [min, max] instructions	Fail time (s) [min,max]
ALU	2689	100	0	[1,5]	[20,120]
HDU	63	100	0	[1,5]	[20,90]
Decoder	1790	75.14	24.86	[1,5]	[20,70]
Forwarding unit	786	91.86	8.14	[1,5]	[32,338]
Register File	1514	100	0	[1,5]	[20,120]
ID-EX register	710	97.75	2.25	[1,5]	[31,830]
EX-MEM register	199	100	0	[1,5]	[29,62]
MEM-WB register	308	100	0	[1,5]	[23,42]

Simulation-Based fault propagation analysis

In the analysis of faults within the RV32IMCZicsr processor components, fault simulation with a random pattern approach was employed. This involved the use of a RISC-V Instruction Generator, responsible for generating random instructions to serve as test stimuli for the analysis. For test purposes, the Instruction Generator constrained the instructions to comply with the

RV32IMCZicsr ISA and processor starts in a clean state. Figure 9.12 presents the fault coverage of processor components when exposed to varying test input lengths, specifically 100, 150, 200, 250, and 300 instructions. As depicted in the figure, the fault coverage for all components is notably lower in comparison to the exhaustive technique.

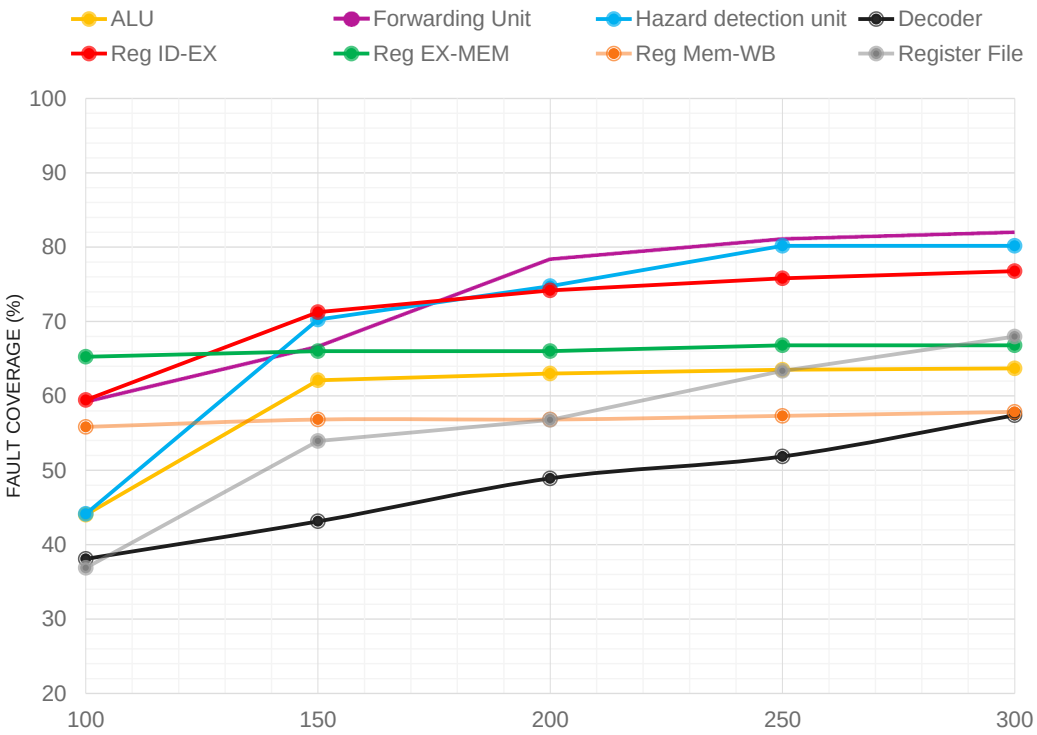


Figure 9.12: Simulation-based fault propagation analysis

In the case of the pipeline registers, the fault coverage remains relatively consistent across different test lengths. Conversely, a consistent rise in fault coverage is evident for the Decoder and Register File. This can be attributed to the larger test inputs encompassing a greater number of instructions, consequently activating more faults within the Decoder and Register File. Notably, achieving fault sensitization for these components necessitates specific instructions. For instance, the initial 100 instructions may lack certain instruction classes, potentially failing to sensitize particular faults associated with the Decoder's branch instruction decoding. Therefore, longer test sequences accommodate a broader range of instruction classes, enabling more registers to be read/written and enhancing fault coverage accordingly.

ATPG-Based fault propagation analysis

Prior to testing, scan chain insertion was performed using Design Compiler®by Synopsys, with Tessent®by Siemens EDA serving as the testing tool. Table 9.13 provides a comprehensive overview of all fault propagation analysis results obtained through ATPG. It is important to highlight that some effort was invested in constraining the inputs to comply with the RVC32IMC ISA. This involved the removal of illegal instructions that could potentially yield incorrect fault coverage results.

The testing tool classifies the faults as follows:

Table 9.13: ATPG-based fault propagation analysis

Component	% DS	% TI	% BL	% RE	% AU
ALU	97.36	1.76	0.88	-	-
HDU	97.52	2.48	-	-	-
Decoder	77.79	9.41	5.16	2.11	5.53
Forwarding unit	89.59	1.92	0.32	8.17	-
Register File	78.36	1.7	13.3	-	6.64
ID-EX register	80.6	0.6	8.2	-	10.6
EX-MEM register	86.7	1	8.2	-	4.1
MEM-WB register	86.7	0.7	8.4	-	4.2

- DS: Detected faults identified through simulation.
- TI: Tied faults, where the fault value remains constant.
- BL: Blocked faults, indicating that the fault path is obstructed by tied logic.
- RE: Redundant faults, signifying undetectable faults due to inherent redundancy.
- AU: ATPG untestable faults, which are faults deemed untestable due to input (pin) constraints or restricted chain depth

During the comparative analysis of fault propagation, the ATPG approach was employed to evaluate the percentage of faults classified as redundant or detectable. The results demonstrate that the formal-based approach yields precise results, aligning closely with those obtained through the ATPG-based technique for combinational designs. However, it's important to note that the formal technique does not account for tied or blocked logic due to its functional nature. The Forwarding Unit exhibits identical redundancy between the two techniques, while some discrepancies emerge for the Decoder. This disparity arises from the challenge of introducing functional constraints into ATPG, which entails assuming specific behaviors at particular timepoints. As a consequence, the introduction of ATPG constraints results in some faults being categorized as AU, whereas the Decoder's comparison logic classifies these same faults as TI or BL. Nevertheless, it's noteworthy that certain faults in sequential components cannot be analyzed using ATPG, primarily due to faults within the scan chain itself.

9.4.4 Discussions and Observations

This research highlights the effectiveness of the formal-based safety verification method in detecting functional bugs in design hardening mechanisms, without requiring in-depth white-box design knowledge. Additionally, the formal-based fault propagation analysis technique provides accurate results, making it a good alternative to simulation-based methods and similar to the ATPG-based approach for combinational designs. This technique also avoids the problem of fault coverage loss in sequential designs since it does not require scan chain insertion. Overall, this method is a helpful addition to the RTL generation process as it provides essential information on the generated design's redundancy.

9.5 Analysis of the Automated SBST Generation

9.5.1 Experimental Setup

Chapter 8 described the automated flow of generating SBST targetting RISC-V processor cores. The SBST flow was further enhanced with the the fault detection mechanism, namely PFC. To analyze the effectiveness of the methodology, various components of a RV32IMC RISC-V CPU were tested. The components include combinational components such as ALU, HDU, and Decoder, plus sequential components such as Register File and the pipeline register MEM-WB. OneSpin 360 DV-Verify® by SiemensEDA was employed as a formal tool to generate counterexamples, operating on an 11th Gen Intel® Core™ i7-1185G7 @3.0GHz processor with 32 GB of RAM. Fault simulation is performed via the Xcelium® simulator. The primary development environment employed is a 64-bit version of RHEL 7.

9.5.2 SBST results

Table 9.14 displays the statistics of the SBST including the number of faults, the number of patterns, the number of program instructions, the fault coverage (FC), the PFC fault detection rate (FDR) and the total test generation time.

Table 9.14: SBST results on RV32IMC RISC-V CPU

Component	Results					
	# Faults	# Patterns	# Instructions	FC(%)	PFC FDR(%)	Generation time (h)
HDU	111	19	266	91.89	100	0.61
ALU	4732	313	4382	97.63	100	40.5
Register File	11747	492	6888	96.86	100	13.6
Decoder	3224	104	1456	76.3	92.19	38.87
MEM-WB	410	10	140	54.8	100	8.5

HDU testing

The HDU has 111 identified faults. The testing process created 19 different patterns and used them to successfully detect 104 (FC 91.89%) of these faults in approximately 0.61 hours. The other faults did not show any failure example within the allotted time frame and were therefore classified as *undetectable*. Upon closer examination, these faults were deemed untestable. For example, a stuck-at-0 fault cannot be tested on a signal that is tied to a low signal.

ALU testing

The flows generated 313 patterns which identified 4620 faults over the course of approximately 13.6 hours. However, 112 faults could not be detected within the 2-minute limit set for each test. This resulted in a fault coverage of 97.63%. The PFC successfully detected all the faults that occurred because it was connected to the ALU's outputs. It effectively monitored all the results by hashing them and this value was then compared to a predetermined fault-free reference value.

Register File testing

The Register File, a significant sequential part of the system, has 11747 identified faults. To detect these faults, the SBST created 492 patterns that successfully identified 11379 of them over a period of around 40.5 hours. Within the allotted 5-minute timeframe for each test in Onespin, 368 faults could not be detected and were therefore considered unresolved. This results in a fault detection rate of 96.87%. It is important to note that the component's outputs outputs were utilized as strobes. This particular design decision was influenced by the fact that the Register File is prone to a substantial quantity of faults. Many of these faults are potentially functionally undetectable, which, if not managed properly, could result in excessively prolonged test generation times. The approach for finding these faults included using two PFCs that analyzed the Register File outputs which effectively caught all the errors.

Decoder testing

The decoder contains 3224 faults. According to the experimental findings in Table 9.11 the Decoder has around 22.02% redundant faults which was a result of a flaw in the generation framework. Given this fault redundancy, a longer generation time was anticipated because the system is designed to spend the entire timeout period trying to detect each undetectable fault. Therefore 710 faults would be undetectable. With the detection process set to a two-minute timeout for each fault, the system would spend roughly 23.7 hours attempting to identify these undetectable faults. The total time taken for fault generation was about 39 hours, suggesting that the process could be made more efficient. The fault coverage remained stable at 76.3%. The PFC managed to identify about 92.19% of the faults. The remaining 7.8% of the faults are unlikely to affect the CPU's output because they do not influence the Register File or the ALU.

MEM-WB testing

The MEM-WB pipeline register, a compact sequential component, contains approximately 410 faults. In an effort to test for 225 of these faults, which are functional in nature, the system produced 10 test patterns. This testing process took a total of 8.5 hours. Furthermore, the ALU PFC proved to be highly effective while detecting all the functional faults.

Chapter 10

Summary of Contributions

This thesis introduced a set of innovative solutions that address many challenges in the domain of safety verification. The effectiveness and applicability of these proposed solutions along with their key findings have been previously published in several scientific conferences [30, 79, 74, 76, 64, 78, 80, 82]. Furthermore, all of the introduced solutions adhere to the principles of Model-Driven Architecture and are fully automated. The practical value of these techniques has been demonstrated through their successful application to a number of industrial designs within Infineon Technologies AG. A short summary of the contributions is presented in the following.

In this thesis, a generic framework for fault injection handling has been developed based on the in-house generation framework Metagen. A key challenge in the execution of fault injection campaigns lies in the manual efforts required for the setup and provision of input for the fault injection environment. The framework addresses this challenge by fully automating the fault injection process, thereby bridging the gap between specifications and the injection process [74]. As with other Metagen-based frameworks, the core element of this framework is the metamodel, named MetaFI. This metamodel is responsible for defining all fault injection attributes, including fault lists, injection campaign types, and fault analysis elements. The framework generates fault injection testbenches in two distinct views, namely SystemVerilog and Verilator-based C++, thus allowing for the utilization of both commercial and open-source tools. Additionally, a generic documentation generation flow, MetaDOCU, has been developed within the scope of this thesis. One of the primary functions of MetaDOCU is to translate all fault injection attributes into structured documentation views, such as Portable Document Format (PDF).

Fault injection campaigns employ various techniques, including hardware-based, software-based, simulation-based, and emulation-based methods, each with its own advantages and drawbacks. Among these, simulation-based fault injection is commonly used due to its cost-effectiveness. This thesis proposes an approach to fault simulation, utilizing mixed granularity models [79]. Specifically, design components subject to fault injection are represented at a fine-grained gate-level granularity, while the remainder of the design is represented in its original RTL form. This approach contributes to the state-of-the-art techniques by improving the overall fault simulation performance while maintaining sufficient accuracy by injecting faults at the gate-level representation. To achieve this, the RTL generation framework, MetaRTL, and its features are heavily utilized. In this approach, the gate-level netlist of the design is transformed into the MetaRTL-based Model-of-Design (MoD), creating a gate-level MoD. Subsequently,

a series of model transformations are applied, combining the gate-level MoD with the original MoD to form a mixed-granularity MoD. Fault injectors are then inserted into the MoD to enable fault injection. The MetaRTL flow is again utilized to generate the design on a mixed-granularity fashion, after which an equivalence check is performed between the original RTL design and the mixed-granularity version, formally proving that the transformation process does not introduce any bugs.

Another major contribution of this thesis is a novel and fully automated fault emulation framework [76]. The framework enhances observability and controllability of injected faults while improving the performance of the fault injection process. Emulation-based approaches for conducting fault campaigns face the well-known limitation of insufficient I/O ports. To combat this issue, a novel design architecture is proposed, which significantly reduces manual efforts and enables the emulation of an entire CPU subsystem. The fault emulation framework, built upon the model-driven RTL generation framework, MetaRTL, provides a new architecture for fault emulation using FPGAs. The fault handling framework, specifies the necessary architecture and outlines various fault injection campaigns. The fault emulator architecture consists of four main components: the Fault Controller, the Design-under-Test, the Postprocessing Block, and the Data Harvesting Logic. The Fault Controller manages the fault injection campaign; the Postprocessing Block captures emulation traces, stores design's outputs value within a Block RAM and classifies faults according to their effect; the Data Harvesting Logic transmits the classified faults to the Host PC for further analysis. Additionally, the framework includes an on-the-fly analysis technique for fault emulation deriving from Lockstep principles, which effectively manages FPGA memory bottlenecks.

A key contribution of the thesis is the combination of the model-driven approach with formal techniques to analyze the impact of the faults in processor designs. This technique enables the verification of design hardening mechanisms such as ECC, TMR, DMR and delivers fault analysis outcomes without the need for additional manual effort or white-box design knowledge. The technique is capable of verifying all error correction and detection mechanisms in the presence of faults, thereby reducing the complexity of formal methods. The approach extends the complete processor verification technique Symbolic State Quick Error Detection (S^2QED) by enabling fault injection via properties. The computation model for safety verification know as Fault Injection S^2QED (FI-S^2QED) verifies the functionality of the hardening techniques and their integration into the design. The principle of this technique is to add extra constraints in the property that inject a single fault into the design. The property should prove that the functional behavior of the design does not change even in the presence of a fault because it should be fixed by the hardening mechanisms.

The final contribution of this thesis is the development of an efficient and automated methodology for generating customized Software-Based Self Test (SBST) for RISC-V processors. The proposed methodology makes use of formal verification techniques, specifically assertion-based property verification, to extract deterministic test patterns, leading to a high fault coverage. Through the integration of formal verification with fault simulation, the number of necessary properties is reduced by eliminating faults detected by the same test pattern, also known as fault dropping. Additionally, the SBST generation is enhanced with a novel Program Flow Checking (PFC) technique that guarantees a high fault detection rate that aligns with various ASILs from the ISO 26262 standard. Consequently, this approach simplifies the generation of tests while also providing high fault detection capabilities, all while maintaining full automation.

The concepts and techniques highlighted in this thesis have proven their applicability and effectiveness through a series of experiments conducted on a variety of RISC-V based industrial designs, as elaborated in Chapter 9. The simulation on mixed-granularity models yielded a significant performance increase compared to a full gate-level simulation. The simulation-based fault injection technique was expanded to analyze fault propagation across different RISC-V processor variants using several benchmark tests. Performance enhancements were further observed in fault injection campaigns due to the application of the fault emulator, providing a performance gain factor of up to 47.57 times. The hardening verification technique discovered integration bugs of these mechanisms in the processor cores and at the same time provided valuable information about design redundancy through a comprehensive fault propagation analysis. Lastly, the SBST technique combined with PFC was implemented to generate test patterns for multiple components of the RISC-V processor. The automated SBST achieved a very high fault coverage without increasing the design area as it removes the need for Design-for-Test infrastructures.

To summarize, the scientific contributions delivered by this thesis can be listed as follows:

- A generic fault handling framework streamlines and automates fault injection campaigns by autonomously defining all their associated attributes.
- A novel automated framework that enables fault simulation on mixed-granularity models enhances the performance of fault simulation, and at the same time is sufficiently accurate for gate-level fault modelling.
- Fault injection is enabled by inserting fault injectors into the design model, achieved via model transformation. Additionally, the framework integrates an automated formal equivalence check to validate the correctness of this transformation.
- A novel and scalable fault emulator architecture has been introduced that increases fault injection performance significantly.
- A formal-based technique can detect all bugs of design hardening mechanisms while simultaneously verifying their correct integration within the design.
- The integration of automated SBST with the PFC provides a low-cost and effective test solution compared to many Design-for-Test techniques. Importantly, this technique aligns with the ISO26262 ASIL standards regarding fault detection rates.
- All the novel solutions introduced in this thesis are fully automated, adhering to MDE principles. Moreover, their practicality and efficiency have been validated through implementation on multiple industrial designs.

Future work

In future, hardware-based and software-based fault injection techniques shall be investigated. A comparative analysis with the techniques introduced in this thesis would provide a broader understanding of their relative performance and potential merit. Additionally, the application of fault emulation and simulation should be expanded to analyze non-core design structures. Furthermore, the applicability of the SBST technique should be explored beyond processors that utilize the RISC-V Instruction Set Architecture (ISA). These studies would potentially offer notable insights into the adaptability of SBST across different processor architectures, thereby enhancing its overall application in the field of fault detection.

Chapter 11

Deutsche Zusammenfassung

Die Halbleiterindustrie erlebt ein schnelles Wachstum, welches den Bedarf an innovativen Entwicklungsmethoden, insbesondere im Bereich des digitalen Designs, erhöht. Trotz der kontinuierlichen Fortschritte im Halbleiterbereich bleibt die Gewährleistung eines korrekten und zuverlässigen Verhaltens eine große Herausforderung, die die Effizienz des Designs beeinträchtigt und zu längeren Projektfristen führt. Neben der Sicherstellung des korrekten Verhaltens müssen die Entwickler auch gewährleisten, dass die Chips bei der Verwendung in Automobilprodukten mit einem vordefinierten Zuverlässigkeitsgrad arbeiten. Folglich hat die Komplexität des Designs und der Verifizierung von anwendungsspezifischen integrierten Schaltungen (ASICs) aufgrund von Sicherheitsüberlegungen zugenommen, die nach der Verabschiedung der Sicherheitsnorm ISO 26262 für sicherheitskritische Systeme im Automobil ein integraler Bestandteil des Entwicklungsprozesses geworden sind. ISO 26262 stellt sicher, dass sich diese Systeme gemäß den geforderten Sicherheitsniveaus verhalten, indem das Risiko gefährlicher Fehlfunktionen gemindert wird. Die Norm empfiehlt die Fehlerinjektion, um sicherheitskritische Systeme zu überprüfen und zu analysieren; dieser Prozess erweist sich jedoch oft als mühsam und fehleranfällig. Um diese Herausforderungen zu bewältigen, sind fortschrittliche und automatisierte Methoden erforderlich. Dementsprechend automatisiert diese Arbeit den Prozess der Sicherheitsüberprüfung und nutzt die Prinzipien der modellgesteuerten Architektur, um die Produktivität, Qualität und Zuverlässigkeit zu verbessern.

Kommerzielle und Open-Source-Tools zur Fehlerinjektion sind in der Regel so konzipiert, dass sie Eingaben wie die Fehlerliste, die Liste der Signalstroboskope und Testfälle akzeptieren. Das manuelle Schreiben von Eingaben für den Fehlerinjektionsprozess kann mühsam und zeitaufwändig sein, so dass eine Automatisierung dieser Aufgabe sehr empfehlenswert ist. Eine Automatisierung des Prozesses würde die Fehlerinjektion rationalisieren und die Effizienz verbessern. Die in dieser Ausarbeitung vorgestellte Arbeit bietet ein automatisiertes und vielseitiges Framework, das entwickelt wurde, um verschiedene Fehlerinjektionskampagnen zu erstellen und gleichzeitig die Lücke zwischen den Fehlerspezifikationen und dem Fehlerinjektionsprozess mit minimalem Aufwand zu schließen. Das Framework bietet eine herstellerunabhängige Lösung, so dass alle Verilog/SystemVerilog-basierten Simulatoren/Emulatoren verwendet werden können. Das Backend des Frameworks basiert auf dem modellgesteuerten Codegenerator-Framework Metagen. Das Framework für die Fehlerbehandlung besteht aus zwei Teilabläufen wie *Fault Injection Handler* und *Fault Injection Documentation*. Das Kernelement des Fault Handlers ist das MetaFI Metamodell, das alle Fehlerinjektionsattributes

spezifiziert, und das Kernelement der Fault Injection Documentation ist das MetaDOCU Metamodell, das alle Attribute für die Dokumentationsgenerierung spezifiziert.

In dieser Arbeit wird ein neuartiger Ansatz vorgeschlagen, der die Fehlersimulation auf einem gemischten RTL-Modell mit "grobkörniger" Granularität durchführt. Der Ansatz überwindet die Herausforderungen der Gate-Level- und RTL-Fehlersimulation, indem er das Beste aus beiden Welten kombiniert. Der Ablauf der Fehlersimulation ermöglicht es, dass nur die Designmodule, die Gegenstand der Fehlerinjektion sind, mit "feinkörniger" Gate-Level Granularität dargestellt werden, während der Rest des Designs mit der ursprünglichen RTL-Granularität dargestellt wird. Das Modell mit gemischter Granularität wird durch Modelltransformation erreicht. Während dieses Prozesses werden formale Äquivalenzprüfungen durchgeführt, um die Korrektheit der Transformation zu gewährleisten. Dieser Ansatz ermöglicht auch die Fehlerinjektion direkt in das Design, wodurch der Overhead, der Einsatz zusätzlicher Tools zur Fehlerinjektion verursacht wurde, reduziert wird. Die Abwesenheit von Fehlern in diesem Prozess wird durch die Äquivalenzprüfung garantiert. Die vorgeschlagene Fehlersimulation verbessert die Leistung der Fehlersimulation auf Gatterebene drastisch und ist gleichzeitig ausreichend genau. Der MetaRTL-RTL-Generierungsfluss rationalisiert den gesamten Arbeitsablauf und macht ihn hochgradig automatisiert und effizient.

Fehleremulationstechniken haben sich als ein Ansatz zur Erleichterung von Fehlerinjektionskampagnen herauskristallisiert. Diese Techniken verbessern die Gesamtproduktivität, indem sie den Zeitaufwand für die Fehlerinjektion reduzieren und gleichzeitig die Kapazität für gründliche Auswertungen aufrechterhalten. Die Emulation arbeitet im Vergleich zu kommerziellen Simulatoren mit deutlich höheren Frequenzen, was zu einer signifikanten Beschleunigung der Laufzeiten für die Fehlerinjektion führt, wobei eine wesentliche Verbesserung bei der Ausführung längerer Eingabesequenzen zu beobachten ist. In dieser Arbeit wurde ein automatisiertes Framework für die Emulation von Fehlern entwickelt, das verbesserte Beobachtbarkeit und Kontrollierbarkeit von injizierten Fehlern mit Leistungsverbesserungen kombiniert. Eine bekannte Einschränkung emulationsbasierter Methoden zur Durchführung von Fehlerkampagnen liegt in der unzureichenden Verfügbarkeit von I/O-Ports. Um dem entgegenzuwirken, wird eine innovative Designarchitektur vorgestellt. Das vorgeschlagene neuartige Fehleremulations Framework reduziert den manuellen Aufwand erheblich und kann skaliert werden, um ein ganzes CPU-Subsystem auf einem FPGA mit LUTs zu emulieren. Das emulationsbasierte Framework zur Fehlerinjektion, d.h. das Fault Emulation Framework, baut auf dem modellgesteuerten RTL-Generierungsframework MetaRTL auf. Das Framework ist vollständig automatisiert, erweitert das bestehende Fault Injection Framework und setzt es ein, indem es eine neuartige Architektur für die Fehleremulation mit FPGAs bereitstellt. Das vorgestellte Metamodell zur Fehlerbehandlung spezifiziert die notwendige Architektur durch die Beschreibung verschiedener Fehlerinjektionskampagnen.

ISO26262 empfiehlt formale Methoden zur Verifizierung der Integrität der sicherheitskritischsten Elemente. In dieser Arbeit wird ein neuartiger Ansatz vorgestellt, der einen modellgetriebenen Ansatz mit formalen Techniken kombiniert, um die Auswirkungen von Fehlern in Prozessordesigns systematisch zu analysieren. Der Prozess beginnt mit der Nutzung der modellgetriebenen Strategie zur Erstellung von Designs mit gemischter Granularität. Anschließend wird eine formale Verifikationstechnik für Prozessoren vorgeschlagen, die die Verifikation von Design-Hardening-Mechanismen ermöglicht und ohne zusätzlichen Aufwand Ergebnisse der Fehleranalyse liefert. Die Technik ermöglicht die Verifikation aller Fehlerkorrektur- und -

erkennungsmechanismen in Gegenwart von Fehlern ohne zusätzlichen manuellen Aufwand und ohne White-Box-Design-Wissen. Der bestehende Fluss der Generierung von Modellen mit gemischter Granularität ermöglicht die Modellierung von Fehlern auf Gatterebene nur für die vorgesehenen Designkomponenten, wodurch die Komplexität formaler Methoden reduziert wird. Der Ansatz der Fehlerfortpflanzungsanalyse leitet funktionale Designeinschränkungen ab und erreicht eine vergleichbare oder bessere Fehlerabdeckung im Vergleich zu ATPG- oder simulationsbasierten Methoden, und bei gleichzeitiger Beibehaltung der Fehlermodellierung auf Gatterebene. Bemerkenswert ist, dass diese Technik sogar ATPG-untestbare Fehler klassifiziert, die von Einschränkungen abgeleitet sind. Dem modellbasierten Ansatz folgend, ist die in dieser Arbeit vorgestellte Technik vollständig automatisiert und erfordert nur minimale menschliche Eingriffe, die sich hauptsächlich auf die Parameterkonfiguration konzentrieren. Der Prozess der Fehleranalyse arbeitet synergetisch mit der Designgenerierung zusammen und liefert wertvolle Erkenntnisse über redundant generierte Module und/oder Untermodule.

Wie bereits erwähnt, erfordert ein effektives Testen die Einführung zusätzlicher Mechanismen wie z.B. Scan-Ketten. Scan-Ketten erleichtern die Übertragung von Testdaten durch das gesamte Design und ermöglichen so das Testen interner Schaltkreise. Durch die Verbindung von internen Designregistern/Flip-Flops über Scan-Ketten wird die Beobachtung und Kontrolle von unzugänglichen internen Designknoten möglich. Dieser Ansatz stellt sicher, dass nur die kombinatorische Logik zwischen Registern/Flip-Flops getestet werden muss. Nach der Integration von Scan-Ketten in das Design wird das Testen mit Hilfe von Testvektoren durchgeführt, die typischerweise in automatischen Testeinrichtungen (ATEs) gespeichert sind. Eine weitere weit verbreitete DFT-Infrastruktur ist der eingebaute Selbsttest (BIST), bei dem eine eigenständige Testschaltung in das Design integriert wird. BIST-Techniken werden üblicherweise als Memory-BIST (MBIST) zum Testen von Speicherschaltungen und Logic-BIST (LBIST) zum Testen von logischen Schaltungen kategorisiert. Die Implementierung verschiedener DFT-Infrastrukturen führt zu einem erheblichen Overhead in Bezug auf den Bereich und die Leistung. Als Reaktion auf die Herausforderungen beim Testen von Prozessorkernen hat sich eine alternative Technik entwickelt, die als Software-basierter Selbsttest (SBST) bekannt ist. Diese Methode beinhaltet das Testen von Prozessorkernen durch Befehle und bietet eine praktikable Alternative zu hardwarezentrierten Lösungen wie BIST. Im Gegensatz zu hardwarebasierten Selbsttests, die die Aktivierung des nicht-funktionalen BIST-Modus erfordern, kann SBST während des normalen Betriebsmodus des Prozessors durchgeführt werden, ohne dass Designänderungen oder der Einbau zusätzlicher Hardwarestrukturen erforderlich sind. In dieser Arbeit wird ein automatisierter und effizienter Ansatz zur Erstellung von SBST für RISC-V Prozessorkerne vorgestellt, der alle oben genannten Herausforderungen adressiert. Die Extraktion von deterministischen Testmustern durch formale Eigenschaften führt zu einer hohen Fehlerabdeckung, wodurch die Herausforderung, eine hohe Fehlerabdeckung zu erreichen, effektiv angegangen wird. Die Integration der formalen Verifikation mit der Fehlersimulation reduziert die Anzahl der erforderlichen Eigenschaften, indem Fehler, die durch dasselbe Testmuster erkannt werden, eliminiert werden, wodurch die Komplexität der Testerstellung verringert wird. Darüber hinaus wurde die Vielseitigkeit der Technik durch die erfolgreiche Anwendung auf verschiedene Komponenten einer RISC-V-CPU demonstriert, wodurch die vollständige Skalierbarkeit nachgewiesen und die Herausforderung der Skalierbarkeit auf großen Designs überwunden wurde. Beachtenswerterweise ist der Ansatz vollständig automatisiert. Darüber hinaus wird die SBST-Generierung um eine neuartige PFC-Technik erweitert, die

eine hohe Fehlererkennungsrate gewährleistet, die auf verschiedene ASILs des ISO 26262-Standards abgestimmt ist. Folglich erleichtert der vorgeschlagene Ansatz nicht nur die Testerstellung, sondern liefert gleichzeitig eine hohe Fehlererkennung bei voller Automatisierung.

Fazit

In dieser Arbeit wurde eine Reihe innovativer Lösungen vorgestellt, die viele Herausforderungen im Bereich der Sicherheitsüberprüfung angehen. Die Effektivität und Anwendbarkeit dieser vorgeschlagenen Lösungen sowie ihre wichtigsten Ergebnisse wurden bereits in mehreren wissenschaftlichen Konferenzen veröffentlicht. Alle vorgestellten Lösungen folgen den Prinzipien der modellgetriebenen Architektur und sind vollständig automatisiert. Der praktische Wert dieser Techniken wurde durch ihre erfolgreiche Anwendung auf eine Reihe von industriellen Designs innerhalb Infineon Technologies AG demonstriert.

Bibliography

[1] Aurix™ 32-bit microcontrollers for automotive and industrial applications. https://www.infineon.com/dgdl/Infineon-TriCore_Family_BR-ProductBrochure-v01_00-EN.pdf?fileId=5546d4625d5945ed015dc81f47b436c7. Accessed: 2023-07-21.

[2] Functional safety of electrical/electronic/programmable electronic safety-related systems. https://webstore.iec.ch/preview/info_iec61508-1%7Bed2.0%7Db.pdf. Accessed: 2023-07-21.

[3] An introduction to the safety standard iec 61508. http://homepages.cs.ncl.ac.uk/felix.redmill/publications/4B.IEC%2061508%20Intro.pdf. Accessed: 2023-07-21.

[4] Mda-y-chart. https://research.linagora.com/pages/viewpage.action?pageId=3639295. Accessed: 2023-07-14.

[5] Omg group. https://www.omg.org/. Accessed: 2023-07-14.

[6] Part 12: The 2022 wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2023/01/09/part-12-the-2020-wilson-research-group-functional-verification-study-2/. Accessed: 2023-09-13.

[7] Part 7: The 2022 wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2022/11/28/part-7-the-2022-wilson-research-group-functional-verification-study/. Accessed: 2023-09-13.

[8] Part 8: The 2022 wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/. Accessed: 2023-09-13.

[9] Partial reconfiguration. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/partial-reconfiguration.html#:~:text=Partial%20reconfiguration%20(PR)%20allows%20you,FPGA%20design%20continues%20to%20function. Accessed: 2023-09-18.

[10] The semiconductor decade: A trillion-dollar industry. https://www.mckinsey.com/industries/semiconductors/our-insights/the-semiconductor-decade-a-trillion-dollar-industry. Accessed: 2023-09-13.

[11] Chapter 1 - introduction. In *Architecture Design for Soft Errors*, S. Mukherjee, Ed. Morgan Kaufmann, Burlington, 2008, pp. 1–41.

[12] ISO 26262: Road vehicles - functional safety. International Organization for Standardization, 2018.

[13] ABATE, F., STERPONE, L., LISBOA, C. A., CARRO, L., AND VIOLANTE, M. New techniques for improving the performance of the lockstep architecture for sees mitigation in fpga embedded processors. *IEEE Transactions on Nuclear Science 56*, 4 (2009), 1992–2000.

[14] ABRAMOVICI, M., BREUER, M. A., AND FRIEDMAN, A. D. *Design for Testability*. 1990, pp. 343–419.

[15] ALIZADEH, B., AND FUJITA, M. Guided gate-level atpg for sequential circuits using a high-level test generation approach. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2010), pp. 425–430.

[16] ANDRÉS, D., RUIZ, J., GIL, D., AND GIL-VICENTE, P. Fades: A fault emulation tool for fast dependability assessment. pp. 221 – 228.

[17] ANDRÉS, D., RUIZ, J., GIL, D., AND GIL-VICENTE, P. Fault emulation for dependability evaluation of vlsi systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on 16* (05 2008), 422 – 431.

[18] ANTONI, L., LEVEUGLE, R., AND FEHER, B. Using run-time reconfiguration for fault injection in hardware prototypes. In *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems* (2000), pp. 405–413.

[19] ANTONI, L., LEVEUGLE, R., AND FEHÉR, B. Using run-time reconfiguration for fault injection. *Instrumentation and Measurement, IEEE Transactions on 52* (11 2003), 1468 – 1473.

[20] ARLAT, J., CROUZET, Y., AND LAPRIE, J.-C. Fault injection for dependability validation of fault-tolerant computing systems. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers* (1989), pp. 348–355.

[21] ARORA, D., RAVI, S., RAGHUNATHAN, A., AND JHA, N. K. Hardware-assisted runtime monitoring for secure program execution on embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14*, 12 (2006), 1295–1308.

[22] ASENOV, A. Random dopant induced threshold voltage lowering and fluctuations in sub-0.1 /spl mu/m mosfet's: A 3-d "atomistic" simulation study. *IEEE Transactions on Electron Devices 45*, 12 (1998), 2505–2513.

[23] AUGUSTO DA SILVA, F., BAGBABA, A. C., HAMDIOUI, S., AND SAUER, C. Combining fault analysis technologies for iso26262 functional safety verification. In *2019 IEEE 28th Asian Test Symposium (ATS)* (2019), pp. 129–1295.

[24] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 11–33.

[25] BAGBABA, A. C., JENIHHIN, M., UBAR, R., AND SAUER, C. 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)* (2020), pp. 1–6.

[26] BARAZA, J., GRACIA, J., GIL, D., AND GIL, P. Improvement of fault injection techniques based on vhdl code modification. In *Tenth IEEE International High-Level Design Validation and Test Workshop, 2005.* (2005), pp. 19–26.

[27] BARAZA, J.-C., GRACIA, J., BLANC, S., GIL, D., AND GIL, P.-J. Enhancement of fault injection techniques based on the modification of vhdl code. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 6 (2008), 693–706.

[28] BAUDRY, B., GHOSH, S., FLEUREY, F., FRANCE, R., LE TRAON, Y., AND MOTTU, J.-M. Barriers to systematic model transformation testing. *Commun. ACM 53*, 6 (jun 2010), 139–143.

[29] BAUMANN, R. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability 5*, 3 (2005), 305–316.

[30] BAVACHE, V. B., HAN, Z., HARTLIEB, H., KAJA, E., DEVARAJEGOWDA, K., AND ECKER, W. Automated soc hardening with model transformation. In *2020 17th Biennial Baltic Electronics Conference (BEC)* (2020), pp. 1–6.

[31] BAYAR, S., AND YURDAKUL, A. Dynamic partial self-reconfiguration on spartan-iii fpgas via a parallel configuration access port (pcap).

[32] BENSO, A., AND PRINETTO, P. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[33] BENWARE, B., SCHUERMYER, C., TAMARAPALLI, N., TSAI, K.-H., RANGANATHAN, S., MADGE, R., RAJSKI, J., AND KRISHNAMURTHY, P. Impact of multiple-detect test patterns on product quality. In *International Test Conference, 2003. Proceedings. ITC 2003.* (2003), vol. 1, pp. 1031–1040.

[34] BIERE, A., CIMATTI, A., CLARKE, E. M., FUJITA, M., AND ZHU, Y. Symbolic model checking using sat procedures instead of bdds. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)* (1999), pp. 317–320.

[35] BOX, G., AND DRAPER, N. Empirical model-building and response surfaces. wiley series in probability and mathematical statistics.

[36] BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. Symbolic model checking: 10/sup 20/ states and beyond. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science* (1990), pp. 428–439.

[37] BÉZIVIN, J. In search of a basic principle for model driven engineering. *Novatica/Upgrade 5* (01 2004).

[38] CHASE, D. Class of algorithms for decoding block codes with channel measurement information. *IEEE Transactions on Information Theory 18*, 1 (1972), 170–182.

[39] CHEN, L., AND DEY, S. Software-based self-testing methodology for processor cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20*, 3 (2001), 369–380.

[40] CHEN, L., RAVI, S., RAGHUNATHAN, A., AND DEY, S. A scalable software-based self-test methodology for programmable processors. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)* (2003), pp. 548–553.

[41] CHO, H., MIRKHANI, S., CHER, C.-Y., ABRAHAM, J. A., AND MITRA, S. Quantitative evaluation of soft error injection techniques for robust system design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2013), pp. 1–10.

[42] CLARKE, E. M., EMERSON, E. A., AND SIFAKIS, J. Model checking: Algorithmic verification and debugging. *Commun. ACM 52*, 11 (Nov 2009), 74–84.

[43] CONSTANTINESCU, C. Trends and challenges in vlsi circuit reliability. *IEEE Micro 23*, 4 (2003), 14–19.

[44] CORNO, F., CUMANI, G., SONZA REORDA, M., AND SQUILLERO, G. An rt-level fault model with high gate level correlation. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (Cat. No.PR00786)* (2000), pp. 3–8.

[45] COSTA, D., NÓBREGA, L., AND NUNES, N. An mda approach for generating web interfaces with uml concurtasktrees and canonical abstract prototypes. vol. 4385, pp. 137–152.

[46] DA SILVA, F. A., CAGRI BAGBABA, A., HAMDIOUI, S., AND SAUER, C. An automated formal-based approach for reducing undetected faults in iso 26262 hardware compliant designs. In *2021 IEEE International Test Conference (ITC)* (2021), pp. 329–333.

[47] DEVARAJEGOWDA, K. *Model-based Generation of Assertions for Pre-silicon Verification*. doctoralthesis, Technische Universität Kaiserslautern, 2021.

[48] DEVARAJEGOWDA, K., AND ECKER, W. Meta-model based automation of properties for pre-silicon verification. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2018), pp. 231–236.

[49] DEVARAJEGOWDA, K., AND ECKER, W. Meta-model based automation of properties for pre-silicon verification. In *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2018), pp. 231–236.

[50] DEVARAJEGOWDA, K., ECKER, W., AND KUNZ, W. How to keep 4-eyes principle in a design and property generation flow. In *MBMV 2019; 22nd Workshop - Methods and Description Languages for Modelling and Verification of Circuits and Systems* (2019), pp. 1–6.

[51] DEVARAJEGOWDA, K., FADIHEH, M. R., SINGH, E., BARRETT, C., MITRA, S., ECKER, W., STOFFEL, D., AND KUNZ, W. Gap-free processor verification by s2qed and property generation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2020), pp. 526–531.

[52] DI CARLO, S., PRINETTO, P., ROLFO, D., AND TROTTA, P. A fault injection methodology and infrastructure for fast single event upsets emulation on xilinx sram-based fpgas. pp. 159–164.

[53] ECKER, W., AND SCHREINER, J. Introducing model-of-things (mot) and model-of-design (mod) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)* (2016), pp. 1–6.

[54] ECKER, W., AND SCHREINER, J. *Metamodeling and Code Generation in the Springer Science+Business Media Dordrecht*. 01 2016, pp. 1–41.

[55] ECKER, W., VELTEN, M., ZAFARI, L., AND GOYAL, A. The metamodeling approach to system level synthesis. pp. 1–2.

[56] ENE, N. C., FERNÁNDEZ, M., AND PINAUD, B. Attributed hierarchical port graphs and applications. *CoRR abs/1802.06492* (2018), 2–19.

[57] ENTRENA, L., GARCIA-VALDERAS, M., FERNANDEZ CARDENAL, R., LINDOSO, A., PORTELA-GARCIA, M., AND ONGIL, C. Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection. *IEEE Transactions on Computers 61* (03 2012), 313–322.

[58] ESPINOSA, J., HERNANDEZ, C., ABELLA, J., DE ANDRES, D., AND RUIZ, J. C. Analysis and rtl correlation of instruction set simulators for automotive microcontroller robustness verification. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), pp. 1–6.

[59] FADIHEH, M. R., URDAHL, J., NUTHAKKI, S. S., MITRA, S., BARRETT, C., STOFFEL, D., AND KUNZ, W. Symbolic quick error detection using symbolic initial state for pre-silicon verification. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), pp. 55–60.

[60] FALLER, T., DELIGIANNIS, N. I., SCHWÖRER, M., REORDA, M. S., AND BECKER, B. Constraint-based automatic sbst generation for risc-v processor families. In *2023 IEEE European Test Symposium (ETS)* (2023), pp. 1–6.

[61] FERRARETTO, D., AND PRAVADELLI, G. Efficient fault injection in qemu. In *2015 16th Latin-American Test Symposium (LATS)* (2015), pp. 1–6.

[62] FIBICH, C., HORAUER, M., AND OBERMAISSER, R. Device- and temperature dependency of systematic fault injection results in artix-7 and ice40 fpgas. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), pp. 1600–1605.

[63] GAI, S., MONTESSORO, P., AND SOMENZI, F. Mozart: a concurrent multilevel simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 7*, 9 (1988), 1005–1016.

[64] GERLIN, N., KAJA, E., VARGAS, F., LU, L., BREITENREITER, A., CHEN, J., ULBRICHT, M., GOMEZ, M., TAHIRAGA, A., PREBECK, S., JENTZSCH, E., KRSTIĆ, M., AND ECKER, W. Bits, flips and riscs. In *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)* (2023), pp. 140–149.

[65] GHAFFARI, F., SAHRAOUI, F., BENKHELIFA, M., GRANADO, B., KACOU, M., AND ROMAIN, O. Fast sram-fpga fault injection platform based on dynamic partial reconfiguration. vol. 2015.

[66] GIZOPOULOS, D., PSARAKIS, M., HATZIMIHAIL, M., MANIATAKOS, M., PASCHALIS, A., RAGHUNATHAN, A., AND RAVI, S. Systematic software-based self-test for pipelined processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 16*, 11 (2008), 1441–1453.

[67] GONZALEZ-PEREZ, C., AND HENDERSON-SELLERS, B. Modelling software development methodologies: A conceptual foundation. *Journal of Systems and Software 80*, 11 (2007), 1778–1796.

[68] GRINSCHGL, J., KRIEG, A., STEGER, C., WEISS, R., BOCK, H., AND HAID, J. Automatic saboteur placement for emulation-based multi-bit fault injection. In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)* (2011), pp. 1–8.

[69] HAN, Z., WANG, D., RUTSCH, G., LI, B., PREBECK, S. S., LOPERA, D. S., DEVARAJEGOWDA, K., AND ECKER, W. Aspect-oriented design automation with model transformation. In *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)* (2021), pp. 1–6.

[70] HAYNE, R., AND JOHNSON, B. Behavioral fault modeling in a vhdl synthesis environment. In *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)* (1999), pp. 333–340.

[71] HENKEL, J., BAUER, L., DUTT, N., GUPTA, P., NASSIF, S., SHAFIQUE, M., TAHOORI, M., AND WEHN, N. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2013), pp. 1–10.

[72] HSUEH, M.-C., TSAI, T., AND IYER, R. Fault injection techniques and tools. *Computer 30*, 4 (1997), 75–82.

[73] JENN, E., ARLAT, J., RIMEN, M., OHLSSON, J., AND KARLSSON, J. Fault injection into vhdl models: the mefisto tool. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing* (1994), pp. 66–75.

[74] KAJA, E., GERLIN, N., BORA, M., DEVARAJEGOWDA, K., STOFFEL, D., KUNZ, W., AND ECKER, W. Metafs: Model-driven fault simulation framework. In *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2022), pp. 1–4.

[75] KAJA, E., GERLIN, N., BORA, M., DEVARAJEGOWDA, K., STOFFEL, D., KUNZ, W., AND ECKER, W. Metafs: Model-driven fault simulation framework. In *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2022), pp. 1–4.

[76] KAJA, E., GERLIN, N., BORA, M., RUTSCH, G., DEVARAJEGOWDA, K., STOFFEL, D., KUNZ, W., AND ECKER, W. Fast and accurate model-driven fpga-based system-level fault emulation. In *2022 IFIP/IEEE 30th International Conference on Very Large Scale Integration (VLSI-SoC)* (2022), pp. 1–6.

[77] KAJA, E., GERLIN, N., KUNZELMANN, R., DEVARAJEGOWDA, K., AND ECKER, W. Modelling peripheral designs using fsm-like notation for complete property set generation. In *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)* (2023), pp. 508–515.

[78] KAJA, E., GERLIN, N., STOFFEL, D., KUNZ, W., AND ECKER, W. Automated thread evaluation of various risc-v alternatives using random instruction generators. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon)* (2023).

[79] KAJA, E., GERLIN, N., VADDEBOINA, M., RIVAS, L., PREBECK, S., HAN, Z., DEVARAJEGOWDA, K., AND ECKER, W. Towards fault simulation at mixed register-transfer/gate-level models. In *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)* (2021), pp. 1–6.

[80] KAJA, E., GERLIN, N., YUN, U., AL HALABI, J., PREBECK, S., STOFFEL, D., KUNZ, W., AND ECKER, W. A statistical and model-driven approach for comprehensive fault propagation analysis of risc-v variants. In *Proceedings of the Design and Verification Conference and Exhibition (DVCon)* (2024).

[81] KAJA, E., GERLIN, N., ZHAO, B., LOPERA, D. S., HALABI, J. A., KHAN, A. S., PREBECK, S., STOFFEL, D., KUNZ, W., AND ECKER, W. An automated exhaustive fault analysis technique guided by processor formal verification methods. In *2024 25th International Symposium on Quality Electronic Design (ISQED)* (2024), pp. 1–8.

[82] KAJA, E., GERLIN, N., ZHAO, B., SANCHEZ LOPERA, D., AL HALABI, J., SHER AZAM, K., , PREBECK, S., STOFFEL, D., KUNZ, W., AND ECKER, W. An

automated exhaustive fault analysis technique guided by processor formal verification methodss. In *25th International Symposium on Quality Electronic Design* (2024).

[83] KAJA, E., LEON, N. O., WERNER, M., ANDREI-TABACARU, B., DEVARAJEGOWDA, K., AND ECKER, W. Extending verilator to enable fault simulation. In *MBMV 2021; 24th Workshop* (2021), pp. 1–6.

[84] KALIORAKIS, M., TSELONIS, S., CHATZIDIMITRIOU, A., FOUTRIS, N., AND GIZOPOULOS, D. Differential fault injection on microarchitectural simulators. In *2015 IEEE International Symposium on Workload Characterization* (2015), pp. 172–182.

[85] KANAWATI, G., KANAWATI, N., AND ABRAHAM, J. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers 44*, 2 (1995), 248–260.

[86] KANAWATI, G., KANAWATI, N., AND ABRAHAM, J. Ferrari: a flexible software-based fault and error injection system. *IEEE Transactions on Computers 44*, 2 (1995), 248–260.

[87] KOCH, N. Transformation techniques in the model-driven development process of uwe. In *Workshop Proceedings of the Sixth International Conference on Web Engineering* (New York, NY, USA, 2006), ICWE '06, Association for Computing Machinery, p. 3–es.

[88] KOCHTE, M., ZOELLIN, C., BARANOWSKI, R., IMHOF, M., WUNDERLICH, H.-J., HATAMI, N., DI CARLO, S., AND PRINETTO, P. Efficient simulation of structural faults for the reliability evaluation at system-level. pp. 3–8.

[89] KOOLI, M., AND DI NATALE, G. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)* (2014), pp. 1–6.

[90] KRANITIS, N., PASCHALIS, A., GIZOPOULOS, D., AND XENOULIS, G. Software-based self-testing of embedded processors. *IEEE Transactions on Computers 54*, 4 (2005), 461–475.

[91] KRAUTZ, U., PFLANZ, M., JACOBI, C., TAST, H., WEBER, K., AND VIERHAUS, H. Evaluating coverage of error detection logic for soft errors using formal methods. In *Proceedings of the Design Automation & Test in Europe Conference* (2006), vol. 1, pp. 1–6.

[92] KRSTIC, A., LAI, W.-C., CHENG, K.-T., CHEN, L., AND DEY, S. Embedded software-based self-test for programmable core-based designs. *IEEE Design & Test of Computers 19*, 4 (2002), 18–27.

[93] KUNZ, W. Verification of digital systems. University lecture, Oct - Feb 2017-2018. [Lecture notes - VDS, from Prof. Wolfgang Kunz at TU Kaiserslautern].

[94] KUPFERSCHMID, S., LEWIS, M., SCHUBERT, T., AND BECKER, B. Incremental pre-processing methods for use in bmc. *Formal Methods in System Design 39* (10 2011), 185–204.

[95] LEE, D., AND NA, J. A novel simulation fault injection method for dependability analysis. *IEEE Design & Test of Computers 26*, 6 (2009), 50–61.

[96] LEGAT, U., BIASIZZO, A., AND NOVAK, F. Automated seu fault emulation using partial fpga reconfiguration. In *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems* (2010), pp. 24–27.

[97] LENTZ, K., AND HOMER, J. Handling behavioral components in multi-level concurrent fault simulation. In *Proceedings 33rd Annual Simulation Symposium (SS 2000)* (2000), pp. 149–156.

[98] LEVEUGLE, R., CALVEZ, A., MAISTRI, P., AND VANHAUWAERT, P. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition* (2009), pp. 502–506.

[99] LIDEN, P., DAHLGREN, P., JOHANSSON, R., AND KARLSSON, J. On latching probability of particle induced transients in combinational networks. In *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing* (1994), pp. 340–349.

[100] LINGAPPAN, L., AND JHA, N. K. Satisfiability-based automatic test program generation and design for testability for microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15*, 5 (2007), 518–530.

[101] LOPERA, D. S., KUNZELMANN, R. N., KAJA, E., AND ECKER, W. Fake timer: An engine for accurate timing estimation in register transfer level designs. In *2024 25th International Symposium on Quality Electronic Design (ISQED)* (2024), pp. 1–8.

[102] LOPEZ-ONGIL, C., GARCIA-VALDERAS, M., PORTELA-GARCIA, M., AND ENTRENA, L. Autonomous fault emulation: A new fpga-based acceleration system for hardness evaluation. *IEEE Transactions on Nuclear Science 54*, 1 (2007), 252–261.

[103] LUONG, G., AND WALKER, D. Test generation for global delay faults. In *Proceedings International Test Conference 1996. Test and Design Validity* (1996), pp. 433–442.

[104] MAIER, P. R., KLEEBERGER, V. B., MÜLLER-GRITSCHNEDER, D., AND SCHLICHTMANN, U. Embedded software reliability testing by unit-level fault injection. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)* (2016), pp. 410–416.

[105] MUELLER-GRITSCHNEDER, D., GREIM, M., AND SCHLICHTMANN, U. Safety evaluation based on virtual prototypes: Fault injection with multi-level processor models. In *2016 International Symposium on Integrated Circuits (ISIC)* (2016), pp. 1–2.

[106] OETJENS, J.-H., BANNOW, N., BECKER, M., BRINGMANN, O., BURGER, A., CHAARI, M., CHAKRABORTY, S., DRECHSLER, R., ECKER, W., GRÜTTNER, K., KRUSE, T., KUZNIK, C., LE, H. M., MAUDERER, A., MÜLLER, W., MÜLLER-GRITSCHNEDER, D., POPPEN, F., POST, H., REITER, S., ROSENSTIEL, W., ROTH, S., SCHLICHTMANN, U., VON SCHWERIN, A., TABACARU, B.-A., AND VIEHL, A. Safety evaluation of automotive electronics using virtual prototypes: State of the art and

research challenges. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)* (2014), pp. 1–6.

[107] PANDEY, S., LIAO, Z., NANDI, S., GUPTA, S., NATARAJAN, S., SINHA, A., SINGH, A., AND CHATTERJEE, A. Sat-atpg generated multi-pattern scan tests for cell internal defects: Coverage analysis for resistive opens and shorts. In *2020 IEEE International Test Conference (ITC)* (2020), pp. 1–10.

[108] PARASYRIS, K., TZIANTZOULIS, G., ANTONOPOULOS, C. D., AND BELLAS, N. Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), pp. 622–629.

[109] PARVATHALA, P., MANEPARAMBIL, K., AND LINDSAY, W. Frits - a microprocessor functional bist method. In *Proceedings. International Test Conference* (2002), pp. 590–598.

[110] PASCHALIS, A., GIZOPOULOS, D., KRANITIS, N., PSARAKIS, M., AND ZORIAN, Y. Deterministic software-based self-testing of embedded processor cores. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001* (2001), pp. 92–96.

[111] PSARAKIS, M., GIZOPOULOS, D., SANCHEZ, E., AND SONZA REORDA, M. Microprocessor software-based self-testing. *IEEE Design & Test of Computers 27*, 3 (2010), 4–19.

[112] QIU, W., AND WALKER, D. An efficient algorithm for finding the k longest testable paths through each gate in a combinational circuit. In *International Test Conference, 2003. Proceedings. ITC 2003.* (2003), vol. 1, pp. 592–601.

[113] RIEFERT, A., CANTORO, R., SAUER, M., SONZA REORDA, M., AND BECKER, B. A flexible framework for the automatic generation of sbst programs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24*, 10 (2016), 3055–3066.

[114] RUANO, O., GARCÍA-HERRERO, F., ARANDA, L., SANCHEZ-MACIAN, A., RODRÍGUEZ, L., AND MAESTRO, J. A. Fault injection emulation for systems in fpgas: Tools, techniques and methodology, a tutorial. *Sensors 21* (02 2021), 1392.

[115] SAU, S., KOOLI, M., DI NATALE, G., BOSIO, A., AND CHAKRABARTI, A. Schifi: Scalable and flexible high performance fpga-based fault injector. In *2016 Conference on Design of Circuits and Integrated Systems (DCIS)* (2016), pp. 1–4.

[116] SAUER, M., CZUTRO, A., SCHUBERT, T., HILLEBRECHT, S., POLIAN, I., AND BECKER, B. Sat-based analysis of sensitisable paths. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems* (2011), pp. 93–98.

[117] SAUER, M., KIM, Y. M., SEOMUN, J., KIM, H.-O., DO, K.-T., CHOI, J. Y., KIM, K. S., MITRA, S., AND BECKER, B. Early-life-failure detection using sat-based atpg. In *2013 IEEE International Test Conference (ITC)* (2013), pp. 1–10.

[118] SCHREINER, J., FINDENIGY, R., AND ECKER, W. Design centric modeling of digital hardware. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)* (2016), pp. 46–52.

[119] SELIGMAN, E., SCHUBERT, T., AND KUMAR, M. V. A. K. *Formal Verification, An Essential Toolkit For Modern VLSI Design*. Morgan Kaufmann Publishers, 2015.

[120] SESHIA, S. A., LI, W., AND MITRA, S. Verification-guided soft error resilience. In *2007 Design, Automation & Test in Europe Conference & Exhibition* (2007), pp. 1–6.

[121] SHEN, J., AND ABRAHAM, J. Native mode functional test generation for processors with applications to self test and design validation. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)* (1998), pp. 990–999.

[122] SHIVAKUMAR, P., KISTLER, M., KECKLER, S., BURGER, D., AND ALVISI, L. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks* (2002), pp. 389–398.

[123] SORIN, D. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[124] SYNOPSYS. VC Formal Apps. `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`. [Online: Accessed on 07.August.2023].

[125] TABACARU, B.-A. On Fault-Effect Analysis at the Virtual-Prototype Abstraction Level. `https://mediatum.ub.tum.de/doc/1471529/1471529.pdf`, 12 2019. [Online: Accessed on 6.August.2023].

[126] THAKER, P., AGRAWAL, V., AND ZAGHLOUL, M. A test evaluation technique for vlsi circuits using register-transfer level fault modeling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 22*, 8 (2003), 1104–1113.

[127] UBAR, R., JÜRIMÄGI, L., ORASSON, E., AND RAIK, J. Fault collapsing in digital circuits using fast fault dominance and equivalence analysis with ssbdds. In *IEEE/IFIP International Conference on Very Large Scale Integration of System-on-Chip* (2015).

[128] UBAR, R., JÜRIMÄGI, L., ORASSON, E., AND RAIK, J. Fault collapsing in digital circuits using fast fault dominance and equivalence analysis with ssbdds. vol. 483, pp. 23–45.

[129] VADDEBOINA, M., KAJA, E., YILMAYER, A., PREBECK, S., AND ECKER, W. Parallel golomb-rice decoder with 8-bit unary decoding for weight compression in tinyml applications. In *2023 26th Euromicro Conference on Digital System Design (DSD)* (2023), pp. 227–234.

[130] VADDEBOINA, M., KAJA, E., YILMAZER, A., GHOSH, U., AND ECKER, W. Pagori:a scalable parallel golomb-rice decoder. In *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)* (2024), pp. 67–72.

[131] VERIFIC. Verific. https://www.verific.com/.

[132] XILINX. Xilinx. https://www.xilinx.com/products/silicon-devices/fpga.html.

[133] YOSYS. Yosys. https://yosyshq.net/yosys/.

[134] ZHANG, Y., REZINE, A., ELES, P., AND PENG, Z. Automatic test program generation for out-of-order superscalar processors. In *2012 IEEE 21st Asian Test Symposium* (2012), pp. 338–343.

[135] ZHANG, Y., ZHANG, F., YANG, B., XU, G., SHAO, B., ZHAO, X., AND REN, K. Persistent fault injection in fpga via bram modification. pp. 1–6.

[136] ZIADE, H., AYOUBI, R., AND VELAZCO, R. A survey on fault injection techniques. *Int. Arab J. Inf. Technol. 1* (01 2004), 171–186.

Endri Kaja

https://www.linkedin.com/in/endri-kaja/

Email: endrikaja1996@gmail.com

Education

- **Rheinland-Pfälzische Technische Universität** — Kaiserslautern, Germany
  Doctoral Candidate - Electrical and Computer Engineering — *Sep 2020 - June 2024*
  Focuses: *Verification of Digital Systems, Safety Verification, Formal Verification, ISO26262*

- **Technische Universität Kaiserslautern** — Kaiserslautern, Germany
  Master of Science - Embedded Systems; GPA: 1.1/1.0 (with distinction) — *Oct 2017 - June 2020*
  Focuses: *Verification of Digital Systems, Computer Architecture, Digital Design, High Level Synthesis and Logic synthesis, Robust Design Systems*

- **Polytechnic University of Tirana** — Tirana, Albania
  Bachelor of Science - Electronic Engineering; GPA: 8.8/10 — *Oct 2014 - June 2017*
  Focuses: *Advanced mathematics and physics, fundamentals of programming and electronics*

Work Experience

- **Infineon Technologies AG** — Munich, Germany
  Senior Verification and Test Engineer - Reusable on chip testing — *Feb 2024 - Currently*
  - **Development of downloadable tests**: Firmware development to test various components in a System-on-Chip.

- **Infineon Technologies AG** — Munich, Germany
  Doctoral Candidate - Advanced methods for model-driven safety analysis and verification — *Sep 2020 - Jan 2024*
  - **Automated fault injection environment development**: Developed a model-based fault injection flow on a mixed RTL/Gate-level representation of the design.
  - **FPGA-based fault emulation framework development**: Developed an automated fault emulation framework based on saboteur fault injection technique.
  - **Formal verification of RISC-V based processors**: Created and developed automated processor verification techniques based on C-S2QED ISA Modeling with Trace Notation for Context Free Property Generation.
  - **Safety verification of RISC-V based processors**: Automated Statistical Fault Injection on multiple processor variants. Moreover, I applied formal-based fault injection techniques to the processor.
  - **Automated Formal Verification of various design hardening techniques**: Formal verification of ECCs, DMR, TMR.

- **Infineon Technologies AG** — Munich, Germany
  Master Thesis - Verification of Nested Exception Handler for RISC-V Based Processors — *Nov 2019 - June 2020*
  - **Automated Formal Verification of a peripheral nested Interrupt Controller**: Generation of properties guided by an FSM-based property generator framework. Completeness checking via OneSpin Gap-Free verification tool.
  - **Automated Formal Verification of a nested Exception Handler**: Developed properties to verify the behavior of the processor in presence of synchronous and asynchronous exceptions.

- **Infineon Technologies AG** — Munich, Germany
  Internship - Design and Verification of different peripherals — *May 2019 - Oct 2019*
  - **Design and Verification of a configurable ECC**: Design and Formal Properties generation of a configurable SEC/DED ECC.
  - **Design and Verification of a configurable CRC**: Design and Formal Properties generation of a configurable CRC.
  - **Formal Verification of a UART peripheral**: Generated properties to verify the transmitter and the receiver of a UART peripheral.
  - **Formal Verification of an AHB to APB bus bridge**: Generated properties to verify the behavior of the bridge guided by an FSM-based property generator.

Academic Experience and Projects

- **Research assistant**: Developed multiple scripts to extract binary descriptions of the compiled C-code.
- **Verification of Digital Systems Lab**: Introduction to SVA/ITL and Verification of multiple designs. Introduction to Completeness Checking and to TIDAL.
- **Embedded Systems Lab**: Developed and deployed a System-on-Chip on a FPGA-based platform.

Technical Skills Summary

- **EDA Tools**: Onespin, Xcelium, JasperGold , Design Compiler, Tessent, Xilinx
- **Programming**: ITL/SVA/Verilog, Python, C++
- **Software Skills**: Unix, Git, Tcl, UML

Soft Skills Summary

- **Public speaking**: Speaker in multiple conferences such as MBMV-21, DFTS-21, VLSI-SOC-22, DFTS-22, DVCON-US-23.
- **Leadership**: Supervised and advised multiple intern and master thesis students.
- **Problem-solving**: Published multiple novel ideas regarding EDA topics.
- **Teamwork**: Part of a research team focusing on EDA topics.
- **Time-management**: Delivered and published my research according to the predefined deadlines.