FAKULTÄT FÜR
ELEKTROTECHNIK UND
INFORMATIONSTECHNIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Dr.-Ing. Thesis

On Fault-Effect Analysis at the Virtual-Prototype Abstraction Level

Bogdan-Andrei Tăbăcaru

Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

On Fault-Effect Analysis at the Virtual-Prototype Abstraction Level

Bogdan-Andrei Tăbăcaru

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr.-Ing. Ralph Brederlow

Prüfer der Dissertation:

1. Hon.-Prof. Dr.-Ing. Wolfgang Ecker
2. Priv.-Doz. Dr.-Ing. Daniel Müller-Gritschneder

Die Dissertation wurde am 04.03.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 19.12.2019 angenommen.

Acknowledgments

I feel privileged to take the time and acknowledge the people who stood by me and helped me during my research and writing this thesis.

To my supervisor, Wolfgang Ecker, I want to extend my heartfelt gratitude for offering me the research topic in the first place and for supporting me throughout the whole process.

I wish to thank my second supervisor, Daniel Müller-Gritschneder, for taking the time to review my work. I appreciate your feedback and support, and I am looking forward to working together again in the future.

I also want to thank my former colleague and manager, Thomas Kruse. Thomas, your patience, and kindness are unmatched, and I can compare your command of English grammar only to the greatest of scholars.

To my work colleagues, Cristiano and Moomen, I want to thank for the time we spent discussing, arguing, and planning the safety-verification flow for Infineon.

I wish to thank my students, Daniel, Karol, and Leandro, who have helped me to implement and verify the fault-injection tools presented in this work.

I want to acknowledge my friends, Elif, Jen, and Teodora, for their support and for proofreading my thesis.

Finally, I wish to thank my parents, Monica and Doru, for motivating me in my time of need and being there for me.

Bogdan-Andrei Tăbăcaru
Munich, March 14, 2020

Abstract

The increase of automation in today's cars has led to the need for safer and more robust hardware systems in the automotive industry. Consequently, the safety standard for automotive applications, ISO 26262, has been adopted in 2011. ISO 26262 requires safety-critical systems to maintain correct functionality in the event of fault occurrence up to 99% of the time. As a result, safety-critical systems require safety verification and must contain safety mechanisms to reduce their failure rate. Safety experts use safety-verification flows to analyze fault effects on safety-critical systems-on-chip (SoCs), and quantify an SoC's fault-tolerance level. Safety-verification flows enhance traditional functional-verification features (e.g., reference models, test-benches) with fault-injection methods, which emulate the behavior of random hardware faults. Based on fault-injection results, the SoC's fault tolerance is improved by designing better safety mechanisms.

ISO 26262 recommends fault injection into register-transfer-level (RTL) and gate-level models. However, these models are available late in a system's development cycle, which increases the cost of safety-mechanism design. Furthermore, RTL and gate-level models have slow simulation speeds, and thus, long simulation runs. Finally, safety verification on RTL and gate-level models requires injection of thousands and even millions of faults (e.g., bit flips, stuck-at faults) in various scenarios, usually by injecting only one fault per simulation. Such scenarios lead to the injection of millions of faults during slow simulations. Besides, time-to-market windows are becoming increasingly shorter, which increases the need for faster and more efficient simulations. To address these issues, hardware developers are migrating safety-verification methods to higher abstraction levels such as virtual prototypes (VPs) developed using SystemC or TLM models.

Because VPs are more abstract than RTL and gate-level models, they benefit from faster simulation speeds and early availability during SoC development. These advantages contribute to fast fault-injection results and reduced cost of safety-mechanism design. However, this higher level of abstraction causes VP-based safety-verification results to correlate poorly with safety-verification results obtained from RTL and gate-level models. Furthermore, fault injection into VPs can lead to the observation of pseudo-failures (i.e., failures without

counterpart on the physical SoC).

The methods presented in this thesis improve the accuracy of safety verification on VPs, avoid pseudo-failures, optimize the fault-verification space of VPs, and accelerate the safety-verification process. To achieve these goals, the thesis introduces a comprehensive safety-verification flow, named SaVer, three VP-based fault-injection methods, and multiple fault-injection optimization approaches. SaVer automatically abstracts gate-level information to the VP abstraction level and performs fast fault-injection simulations using an optimized fault library. The results obtained after applying SaVer on three microprocessors show (i) 100% correlation between VPs and gate-level models, (ii) 33-34-fold reduction of redundant transient faults, and (iii) acceleration of fault-injection simulations compared to RTL and gate level by up to two orders of magnitude.

Contents

Acknowledgments **v**

Abstract **vii**

1. Introduction **1**
1.1. Safety Verification in the Automotive Industry 2
1.2. Safety Verification on Hardware Architectures 3
1.3. Motivation 5
1.3.1. Linking Safety Analysis and Verification 6
1.3.2. Need for Early Safety-Architecture Exploration 7
1.3.3. Optimization of Safety-Verification Campaigns 7
1.4. State of the Art 8
1.4.1. Simulation-Based Fault-Injection Methods 8
1.4.2. Optimization of Fault-Injection Simulations 12
1.5. Contributions of this Thesis 15
1.6. Previous Publications 17
1.7. Structure of this Thesis 17

2. Safety-Verification of Hardware Models **19**
2.1. Virtual Prototypes 19
2.1.1. SystemC 20
2.1.2. TLM 20
2.2. Simulation-Based Fault Injection 22
2.2.1. Fault-Injection Environment 22
2.2.2. Fault-Injection Attributes 23
2.2.3. Fault-Verification Space 24
2.2.4. Fault-Propagation Paths 25
2.2.5. Types of Hardware Simulators 25
2.2.6. Fault Models 27
2.3. Summary 30

3. Generic Fault-Injection Methods for Virtual Prototypes **31**
3.1. Introduction . . . 31
3.2. SCFIT . . . 31
3.2.1. Fault-Injection Locations . . . 32
3.2.2. Fault Models . . . 34
3.2.3. Fault-Injection Flow . . . 34
3.2.4. Model-Based Automation and Graphical User Interface . 35
3.3. Simulator Commands for SystemC . . . 36
3.4. Simulator Commands for SystemC/TLM . . . 38
3.4.1. Injectable Interface . . . 39
3.4.2. Injectable Payload . . . 39
3.4.3. Injectable Sockets . . . 40
3.5. Summary . . . 42

4. Improving the Correlation of Fault-Injection Results **43**
4.1. Introduction . . . 43
4.2. Fault-Masking Effects . . . 43
4.2.1. Electrical . . . 44
4.2.2. Latch Window . . . 44
4.2.3. Temporal . . . 44
4.2.4. Logical . . . 44
4.3. Pseudo-Faults and Pseudo-Failures . . . 45
4.4. Fault-Matching Points . . . 47
4.5. Augmentation of Virtual Prototypes with Gate-Level Data . . . 51
4.5.1. VERITAS . . . 51
4.5.2. VERITAS++ . . . 54
4.6. Summary . . . 56

5. Optimizing Fault-Injection Simulations **59**
5.1. Introduction . . . 59
5.2. Measures for Verification Completeness . . . 60
5.3. SaVer . . . 62
5.3.1. Fault-Injection Flow . . . 62
5.3.2. Fault-Injection Methods . . . 62
5.4. Spatial and Temporal Fault Pruning . . . 63
5.4.1. Removal of Redundant Fault-Injection Locations . . . 63
5.4.2. Simulation-Trace Analysis . . . 64
5.5. Parallelization of Fault-Injection Simulations . . . 65

5.6. Simulation Checkpointing 66
5.6.1. Checkpoint 66
5.6.2. Restore 68
5.6.3. Checkpointing within SaVer 69
5.6.4. Performance Analysis 70
5.7. Summary 72

6. Experimental Results and Discussion **75**
6.1. Application Example 75
6.1.1. Adder Architectures 75
6.1.2. Microprocessor Cores 76
6.2. Experimental Setup 77
6.3. Quantitative Analysis of Fault-Matching Points 78
6.4. Qualitative and Quantitative Analysis of Permanent-Fault Effects 80
6.5. Simulation Performance of Fault-Injection Methods 85
6.5.1. SCFIT 85
6.5.2. Injectable TLM Sockets 86
6.5.3. Fault-Injection Objects 87
6.6. Performance Measurements of Checkpointing Mechanism 87
6.6.1. Requirements of Hard-Disk Space 88
6.6.2. Generation Time of Checkpoints 89
6.7. Reduction of Fault-Verification Space 90
6.7.1. Spatial Fault Pruning 90
6.7.2. Temporal Fault Pruning 90
6.8. Speed-Up of Fault-Effect Analysis 91
6.8.1. VERITAS and VERITAS++ 91
6.8.2. Checkpointing Mechanism 95
6.9. Summary 96

7. Conclusion **99**

A. ISO 26262 and Functional Safety **103**
A.1. Introduction 103
A.2. Hardware Models 103
A.3. Automotive Safety-Integrity Levels (ASILs) 104
A.4. Failures Modes 105
A.5. Tolerance-Time Interval 106
A.6. Safety-Coverage Metrics 108
A.6.1. Single-Point-Fault Metric (SPFM) 108

A.6.2. Latent-Fault Metric (LFM) 108

B. Architecture Vulnerability Factor **109**

C. Linking Safety Analysis to Fault-Injection Frameworks **111**

Acronyms **115**

Glossary **119**

List of Figures **121**

List of Tables **125**

Bibliography **127**

1. Introduction

Car manufacturers currently focus on implementing advanced driver-assistance systems (ADAS) [1] and even autonomous-driving features [2]. However, if not developed correctly, such features can sometimes cause systems-on-chip (SoCs) to fail, which can then lead to injury and even loss of human life [3], [4]. Consequently, in 2011, the automotive industry has adopted the safety standard ISO 26262 [5] to regulate the development, verification, and validation of safety-critical SoCs against random hardware (HW) faults.

One of the leading causes of system failures in the automotive industry is the increasing density of electronic and electric devices inside road vehicles. As cars become more automated, the number of safety-critical SoCs within a car also increases. Furthermore, these SoCs process more and more real-life data, such as monitoring car and pedestrian activities and analyzing traffic signs. These increases in density and data-processing rate make SoCs more susceptible to the effects of random HW faults. As a result, fault occurrences can cause data corruption and data loss, which then can lead to system failures. This behavior prompts the need for more robust and safer SoCs.

Another cause of system failures is the transistor size. Currently, HW devices continue to decrease in size as described by "Moore's Law" [6]. However, these shrinking technology nodes increase the sensitivity of safety-critical SoCs to the effects of random HW faults, such as cosmic radiation [7] and aging effects [8]–[10]. This behavior further underlines the need for robust and safe SoCs.

To mitigate the effects of random HW faults, various types of fault-mitigation methods, also called safety mechanisms, have been developed. The goal of these components is to harden a SoC and improve its self-healing capabilities [11]. However, safety mechanisms increase the development complexity and cost of safety-critical SoCs, which involves balancing several trade-offs across the system's power consumption, performance, and area characteristics. For instance, the addition of a lock-step mechanism increases the system's area and power consumption, while other mechanisms may reduce its performance [12], [13]. Furthermore, ISO 26262 increases the verification and validation complexity of SoCs by recommending the utilization of safety-analysis and safety-verification

methods to evaluate the benefits of safety mechanisms.

Because of these added constraints, semiconductor companies must improve their process of developing, verifying, and validating SoCs to meet their time-to-market constraints. To achieve this goal, they require methods which speed up and improve the accuracy of safety analysis and safety verification on safety-critical SoCs. Even though many such methods already exist, there is still a growing need for more automated, faster, and more accurate safety analysis and verification.

1.1. Safety Verification in the Automotive Industry

The safety standard for automotive applications, ISO 26262, defines the product life-cycle process (development, verification, and validation) of safety-critical systems (Fig. 1.1). Additionally, ISO 26262 provides requirements for functional safety analysis across this life-cycle (Annex A). As a result, a system's safety level must be analyzed at different stages of a system's development. Safety-analysis methods can be broken down into:

Methods based on expert judgment predict a system's fault tolerance, also called failure-in-time (FIT) rate, using data from field returns of similar systems, pre-compiled safety catalogs (e.g., SN 29500), and expert judgment [5]. The most widely utilized methods in this category are the failure modes, effects, and diagnostics analysis (FMEDA) [14], the fault-tree analysis (FTA) [15], and the dependent-failure analysis (DFA) [16].

Analytical methods use mathematical models to predict a system's fault tolerance. These approaches can be grouped further into methods which calculate a system's architecture-vulnerability factor (AVF) [17]–[26] (Annex B), and those which use graph-based models such as Markov chains, binary-decision diagrams, arithmetic-decision diagrams, and even Petri nets [27]–[30].

Since analytical methods and those based on expert judgment do not analyze the system's behavior, ISO 26262 recommends fault-injection-based safety verification for products classified using the automotive safety-integrity levels (ASILs) C or D. According to the safety standard, fault-injection methods "serve as a check of particular points of the HW design [...] for which analytical methods [...] are not considered to be sufficient" [5]. Faults must be injected at multiple

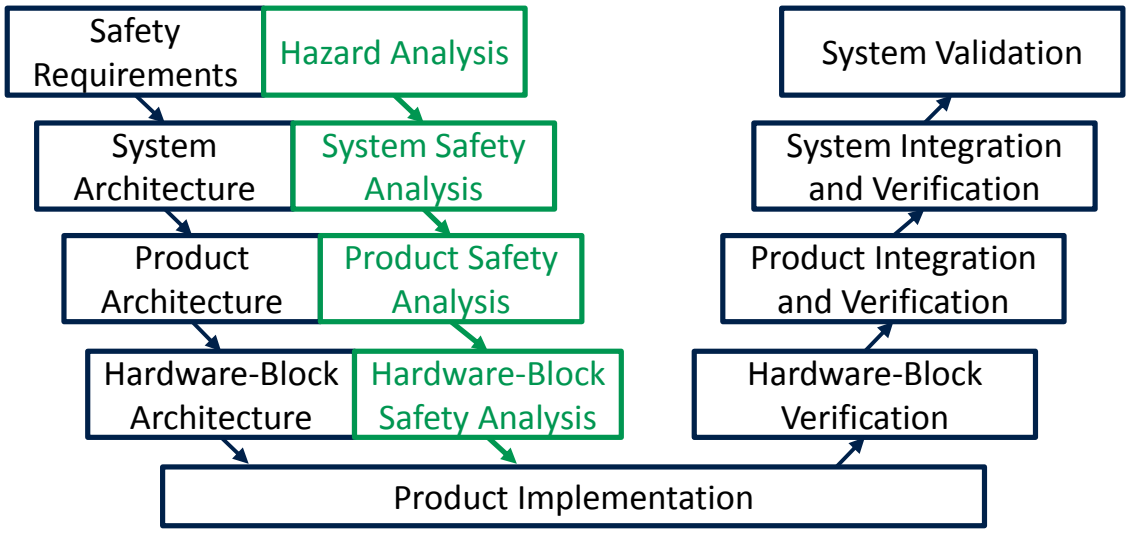


Figure 1.1.: ISO 26262's simplified V-model diagram

stages of the safety standard's V-model (Fig. 1.1). On the left side of the V-model, safety analysis is required at three main stages before product implementation: system, product, and HW block architectures. On the V-model's right side, fault injection is needed to test and validate the analyzed system.

ISO 26262 regards fault injection as a suitable method for validating the safety requirements of a safety-critical system, verifying the effectiveness of safety mechanisms, and estimating the system's fault tolerance. Safety mechanisms are usually verified by injecting HW faults directly into system components using a method called direct fault injection. Additionally, HW faults can be inserted randomly into a system using methods for statistical fault injection (SFI). SFI determines a system's failure distribution, fault vulnerability, and FIT rate. However, ISO 26262 does not specify how to link the results of safety analysis and fault-injection-based safety verification.

1.2. Safety Verification on Hardware Architectures

Currently, the automotive industry is researching more effective methods to develop, verify, and validate safety-critical SoCs. Existing safety-verification methods aim to improve the development of safety mechanisms for SoCs. This goal is achieved by estimating the fault-tolerance level of a safety-critical SoC, and be determining which components of the SoC are most sensitive to HW faults. Once these values have been determined, safety mechanisms can be

applied systematically to the system to improve its fault-tolerance level (FIT rate) and maintain optimal power, area, and performance trade-off. Safety-verification methods can be grouped based on their execution platform as:

Physical testing determines the fault tolerance of safety-critical SoCs by bombarding the physical SoC with radiation [29].

Emulation methods estimate a SoC's fault tolerance by executing a system's gate-level net-list on dedicated HW devices such as field-programmable gate arrays (FPGAs) [31]–[34] or graphics-processing units (GPUs) [35].

Simulation methods estimate a SoC's fault tolerance by simulating the system's functionality at different levels of abstraction on one or more personal computers (PCs) using a HW simulator [29], [36].

While these fault-injection methods can be used successfully to determine a system's fault tolerance, physical testing and emulation methods significantly increase the cost of SoC verification and validation because of their need for dedicated HW components (e.g., radiation sources, FPGAs, GPUs). Additionally, they can only be applied late in the development cycle of a safety-critical SoC once the system's gate-level net-list is available or the system has been already manufactured. Consequently, the SoC may need to be redesigned and even re-manufactured several times before it reaches its required failure rate; a process which can also significantly increase the SoC's development cost.

State-of-the-art simulation methods can be further categorized based on the abstraction level of the HW models they are simulating:

Transistor-level models characterize the fault tolerance of technology cells (e.g., logic gates, flip-flops) using in SPICE simulations [29], [37].

Gate-level net-lists are similar to emulation methods, but executed on one or more PCs instead of FPGAs or GPUs [7], [10], [29], [38]–[45].

Register-transfer-level (RTL) models are more abstract than gate-level net-lists, and thus, offer better simulation performance [40], [41], [45]–[49].

Unfortunately, these abstraction levels suffer from different drawbacks. The transistor abstraction level uses models of currents and voltage potentials, and is too detailed to be used in simulations of entire safety-critical SoCs. Gate-level net-lists and RTL models suffer from slow simulation speed and are available late

in the development phase of a SoC, when changes to the system are expensive and may require redesign. Furthermore, the high granularity of gate-level net-lists and RTL models leads to long and complex safety-verification campaigns (e.g., thousands and even millions of simulations).

Commercial simulation-based fault-injection methods are also available [50]–[54]. However, these methods require considerable manual input and expert judgment. Furthermore, they are applied in the late stages of SoC development on gate-level net-lists and RTL models.

1.3. Motivation

According to ISO 26262, the results of safety analysis on safety-critical SoCs must be proven using fault-injection methods. However, the safety standard only recommends this approach for systems classified as ASIL C or D. Furthermore, it does not provide a method to link the results from fault injection to those from safety analysis. Consequently, these results are mainly linked manually, which is an error-prone approach. Therefore, there is a need for a solution which automates this process, reduces the risk of human error, and bridges the gap between predictions, which are based on expert judgment, and fault-injection results.

As already mentioned, state-of-the-art fault-injection methods can only be applied late in the development cycle of safety-critical SoCs. In this case, systems which do not achieve the desired fault-tolerance level may require a cost-intensive redesign of safety mechanisms. Consequently, there is a need to migrate fault-injection methods to earlier stages of SoC development, for instance at the architecture-exploration stage, when safety mechanisms are also developed.

Apart from their late applicability, state-of-the-art safety-verification campaigns require the execution of a large number of faults to obtain significant statistical results. For large SoCs with various application scenarios, such as a car's engine-control unit (ECU) [55], such campaigns increase the cost of safety verification, and can even become unfeasible. Therefore, there is a need to speed up fault-injection methods, and reduce the size and complexity of safety-verification campaigns.

1.3.1. Linking Safety Analysis and Verification

Based on the recommendations of ISO 26262, safety analysis must be performed on a SoC's architecture to predict its fault tolerance. Given their applicability, safety-analysis methods can be defined as:

Top-down methods (e.g., FTA) start at the system's top-level (e.g., package and pins), and uses deduction to hierarchically determine the SoC's failure causes down to its smallest components (e.g., gates, transistors).

Bottom-up methods (e.g., FMEDA) start with a flattened list of failure causes for the system's smallest components and uses induction to predict the failure rate of the component hierarchies, and finally the entire system.

One way to analyze a system's safety levels is by breaking down a safety-critical SoC into hierarchical sub-components, and determining their root causes using FTA. Next, DFA is applied to the resulting fault tree, which identifies the contributions of dependent failure causes to the system's failure rate. Then, individual failure rates are predicted for the leaf nodes of the fault tree using FMEDA. Finally, the failure rate of the entire system is determined hierarchically. Typical sources for the prediction of failure rates are safety catalogs (e.g., SN 29500), field-return data from existing products, expert judgment, and others. Similar safety-analysis methods already exist such as HiP-HOPS [56].

Using these safety-analysis results, safety mechanisms are developed to increase the SoC's fault tolerance, and reach the system's required ASIL. If the SoC is an ASIL C or D product, safety verification through fault injection is recommended to prove the predicted failure rate of a particular system component. For this reason, the predicted failure rates from safety analysis are used to derive information necessary for fault-injection campaigns, such as where to inject a fault within the SoC (e.g., registers, flip-flops, memory cells), or the type of the injected HW fault (e.g., open or short circuit, or bit-flip). These derived data are stored in so-called fault libraries. Then, the results of fault-injection campaigns are compared with the initial predictions from safety analysis. In the case of a substantial mismatch between the two values, it may be necessary to design more efficient safety mechanisms.

Safety analysis and verification are a continuous process throughout a SoC's development cycle. Thus, as the architecture of a safety-critical SoC is updated, the initial safety analysis must also be refined to reflect the new architecture. However, these results are mainly updated manually, which is an inherently error-prone process. Similarly, faults must regularly be injected (with every update

of the SoC's architecture) as part of the SoC's safety verification. However, the creation and maintenance of fault-injection libraries is usually a manual approach. Moreover, results from fault-injection experiments are annotated manually to the safety analysis.

1.3.2. Need for Early Safety-Architecture Exploration

Before developing a safety-critical SoC using RTL and gate-level models, the feasibility of the SoC's architecture must first be analyzed. Safety analysis uses the resulting SoC architecture to predict its failure rate and derive safety mechanisms which improve the architecture's fault-tolerance level. Currently, this feasibility is analyzed by simulating abstract HW models of the architecture called virtual prototypes (VPs). Depending on the industry where they are applied, VPs may be modeled in many ways (e.g., LabVIEW, Matlab, Simulink [57]). In the automotive industry, the SystemC and transaction-level-modeling (TLM) libraries have become the *de facto* standard for VP modeling [58].

VPs modeled using SystemC and TLM are utilized to explore different architecture configurations of a SoC, such as various bus architectures, or communication interfaces across central-processing units (CPUs) and peripherals. Furthermore, VPs are used to evaluate the configuration's impact on overall system performance, area, and power consumption. For safety-critical SoCs, such VPs can be extended to explore various safety-mechanism configurations as well, and thus, perform early safety-architecture exploration. To enable this capability on VPs, efficient fault-injection methods are required for SystemC and TLM models. However, existing safety verification methods focus on fault injection into RTL models and gate-level net-lists.

1.3.3. Optimization of Safety-Verification Campaigns

As mentioned in section 1.2, emulation and simulation-based fault injection into gate-level net-lists are currently the preferred safety-verification methods. The main benefit of these techniques is their similarity to the physical system. Since such HW models are highly detailed, fault-injection results are accurate and can be reproduced on the physical design. Furthermore, because of these models' increased granularity, faults may be injected into many different parts of a safety-critical SoC, such as registers, combinational logic gates, flip-flops, and others. Such systems may also experience the effects of several types of HW faults such as electromigration, cross-talk, or neutron strikes. As a result, safety-

verification campaigns end up having to evaluate the outcomes of thousands and even millions of fault injections.

The verification space of safety-critical SoCs also increases with the increase of the SoC's complexity, and the real-world applications which such SoCs have to execute. Modern SoCs have different operational modes (e.g., *test mode*, *normal mode*), and also benefit from power-saving features such as *sleep mode* for different unused components, or clock and power gating. For this reason, the SoC's application may influence the propagation of faults through the system. For instance, faults injected into components which are in *sleep mode* do not propagate through the system, and do not produce outcomes. Thus, such faults only waste resources (e.g., verification time). For this reason, safety-verification methods cannot treat systems as black-boxes, which can be verified by purely statistical means. In this case, effective fault-injection methods must also consider the system's application.

To address the challenges of the slow execution speed of RTL and gate-level models, and their inherent late availability during SoC development, optimizations are required to speed-up the execution of fault-injection campaigns, reduce the size of the SoC's verification space, and offer faster safety-verification results.

1.4. State of the Art

This section presents state-of-the-art methods for fault-injection-based safety-verification approaches developed for SystemC and TLM models, and for optimizing fault-injection simulations.

1.4.1. Simulation-Based Fault-Injection Methods

The fault-injection approaches presented in this section are categorized based on the breadth of their applicability: methods only applied to VPs, and methods used across multiple abstraction levels (i.e., VPs, RTL models, and gate-level net-lists).

At the Virtual-Prototype Level

The first fault-injection approaches for SystemC-based VPs have created SystemC models similar to gate-level net-lists and RTL models [59]–[62]. Furthermore, the authors of [61], [62] have implemented traditional design-for-test methods, such as automatic test-pattern generation (ATPG), for such SystemC models. However,

these models have slower performance than commercial HW simulators [62]. In [63], the SystemC library has been enhanced to support concurrent fault simulation with the goal of improving the slow simulation performance of SystemC-based gate-level models. Here, dedicated 32-bit data types have been used to store the results of 31 parallel fault simulations and one reference simulation. The simulator has only been applied to combinational logic.

In [64], the authors have developed a generic debugging approach for SystemC VPs. They have set breakpoints on SystemC data types using the GNU Debugger (GDB) to access data from the HW model, control the SystemC simulation, and manually inject faults. This method allows fault injection without having to modify the original SystemC model (i.e., non-intrusive method). However, this approach does not support automated or generic fault injection into the VPs. Moreover, TLM models are not supported.

In [65], the authors have created fault-injection data types for SystemC models which are controlled from a test-bench during a SystemC simulation. Thus, VPs which use these data types benefit from fault-injection capabilities. However, legacy models must be manually updated, which makes this method intrusive. Furthermore, this approach does not support fault injection into private SystemC data members.

In [66], SystemC models are parsed, analyzed, and transformed into extended finite-state machines (FSMs). During the transformation phase of the SystemC code, this approach uses code mutation to introduce permanent faults into the FSM's implementation. However, this method requires the generation of multiple FSMs to analyze the behavior of different permanent faults, which reduces the method's performance. Furthermore, this approach only supports the injection of a limited set of fault types. Finally, it has reduced simulation performance due to the complex nature of the generated FSMs.

Other approaches combine the SystemC library with other programming languages. One such work links SystemC to Python [67] using the simple wrapper interface generator (SWIG). The authors have named the resulting library SystemPy. SystemC modules are created directly in Python and still use the SystemC simulation kernel implemented in C++. However, SystemPy only supports manual fault injection. Other works, such as ReSP, use the Boost.Python library to connect to the SystemC and TLM library and extend the SystemC simulation kernel [68]–[71]. ReSP can inject faults into SystemC models and perform safety verification using Python. ReSP is non-intrusive and uses Python's introspection capabilities to enhance the SystemC library. However, the extra Python classes and computational complexity add a significant simulation

overhead to the SystemC simulation, which degrades the VP's performance.

The VP-based fault-injection approaches mentioned above focus on manually abstracting RTL and gate-level models to the VP level. However, the authors of [72] have reversed the process by using a high-level logic-synthesis tool to generate RTL code from SystemC VPs. The authors have injected faults using static code mutation. Thus, instead of modifying the values of logic gates during simulation, the authors have modified its function (i.e., arithmetic, logic, shift, relational, and binary). However, the high-level synthesis tool generates registers, which cannot be reached by the fault-injection tool during simulation. Finally, the high-level synthesis tool employed in this approach is used only for a reduced set of HW architectures.

In [73], fault injection is performed using a TLM-based saboteur, which is a TLM module that interconnects a TLM initiator and a TLM target component. Fault injection is broken down into three methods: cycle-accurate models, untimed TLM models, and timed TLM models. In the case of fault injection over multiple abstraction levels, this approach requires manually-implemented adapters across every abstraction level to enable usage of TLM fault-injection modules. This method uses two fault models: data and address faults. Even though this approach implements a robust fault-injection mechanism for TLM models, it suffers from two main drawbacks. First, it requires manual implementation of adapters and fault-injection modules. Second, the re-usability of fault-injection modules is limited to using generic payloads. In the case of TLM models which implement specialized sockets or payloads, new fault-injection modules must be created manually. As a result, this approach does not scale for large safety-critical microcontrollers (e.g., the STMicroelectronics SPC5^{TM} [74] family, the Infineon AURIX^{TM} [55] family).

In [75], faults are injected into TLM models by hijacking the virtual tables of Microsoft's VisualC++ and GNU's GCC compilers. Additional virtual methods are inserted into components of the TLM library during compilation time by creating so-called *hooks* to the compiled TLM models. These hooks are accessed from an ***e*** test-bench [75]. The ***e*** language is an object-oriented verification language for hardware-description languages (HDLs) [76]. It contains aspect-oriented features which enable users to define objects (e.g., classes, enumerations, and other data types) and then, extend these objects' functionality without modifying their already existing implementation. This approach allows third-party models to be fitted for fault injection, even though they initially did not come equipped with fault-injection capabilities. For this reason, the method is non-intrusive. However, the authors have reported a high simulation overhead

of 20-30% introduced by the instrumented virtual tables and interposed modules. Furthermore, over 40 virtual methods have been added to the TLM library, which renders the fault-injection application programming interface (API) complicated to use.

At Mixed Abstraction Levels

In [77], VPs are co-simulated with models from the RTL and the gate level. The authors' goal has been to speed up fault-injection simulations by combining the advantages of two or more abstraction levels, such as the granularity of gate-level models with the verification speed of VPs. Similarly, in [78], VPs are used to quickly bring a CPU into a state ready for fault injection. Afterward, the VP information is used to start a detailed RTL simulation of the same CPU. Then, the RTL models are used to inject faults and observe their effects.

In [79], faults are injected separately into sequential components of VPs, RTL, and gate-level models to quantify the models' accuracy trade-off. The authors have used simulation as well as emulation techniques to assess this trade-off. Several failure outcomes have been recorded as a result of injecting faults into a LEON3 processor. Furthermore, the authors have discovered inconsistencies of fault-injection results across VPs, and RTL and gate-level models.

In [80], [81], the authors have attempted to speed up the fault-simulation process on HW models. For this reason, they have characterized the fault tolerance of a LEON3 processor using RTL models. Afterward, they have used an instruction-set simulator to migrate the fault effects from RTL models to VPs. The authors have abstracted faults manually to the VP level. Similar to [79], the authors have noted different fault-injection results across the abstraction levels.

In [82], a cross-layer framework migrates faults from detailed models to more abstract models. The authors have achieved this by injecting faults into gate-level net-lists. The observed fault effects have then been reproduced at the VP level using statistical methods. Finally, the faults which have produced the same effects as on the gate-level net-lists have been used to characterize the analyzed system's fault tolerance. Faults are injected into TLM-based VPs by modifying TLM-specific components such as the TLM-payload attributes (e.g., address, data pointer, the transaction's delay parameter). Furthermore, faults are transferred from gate-level net-lists to the VP through a manually implemented wrapper function, which maps parts of the gate-level net-list to functional blocks of the VP. Since VPs are more abstract than gate-level net-lists, they have been manually extended to more closely resemble the gate-level implementation.

1.4.2. Optimization of Fault-Injection Simulations

This section presents two types of methods, which optimize the fault-injection-based safety verification. The first type reduces the complexity of fault-injection experiments. The second type speeds up the remaining experiments using simulation checkpointing.

Reduction of Safety-Verification Complexity

As the complexity of modern safety-critical SoCs increases, so do the number of faults which can affect the system, the system's sensitivity to faults, and the complexity of modeling optimal fault-injection scenarios. One of the main problems of safety verification is determining which faults propagate through the system and cause failures, and which get masked (i.e., do not cause failures) [83]. Another significant problem in safety verification are faults which have the same propagation paths, and thus, lead to the safe failures. In [84], [85], two methods are introduced which cluster such faults and exclude them from subsequent simulations: fault pruning and fault collapsing. **Fault collapsing** eliminates faults, which have the same effect on the system. **Fault pruning** removes faults, which are always masked by a system.

In [86], a method is presented to group interconnected combinational gates which are connected to sequential elements (e.g., latches, flip-flops) or output ports. This group is called a cone of influence (COI). As a result, faults occurring within the COI can be collapsed to a single fault occurring within the output or sequential element at the output of the COI. If a COI is connected to multiple outputs, the effects of such a fault injection must be spread out across each output, for instance, using DFA [16].

In [87], a method to perform fault pruning is presented which analyzes the logic dominance of combinational logic gates (i.e., '0' for AND gates, and '1' for OR gates). These dominance values are decided at each simulation step using a reference simulation. If the gate's input has a dominant value, faults are not injected into the other input of that logic gate. In [88], a similar approach is introduced which prunes faults using static code analysis. In this case, safety-critical blocks are broken down into basic blocks. Then, these blocks are analyzed based on available HW masking effects and the definition of constant variables. Faults which are always masked by the system are pruned out. In [20], [52], faults are pruned by analyzing the behavior of safety-critical HW components (e.g., logic gates, flip-flops, latches, memory cells). This behavioral analysis executes gate-level net-lists and RTL models and determines which faults can

propagate through the system and which get masked immediately after their injection.

In [89], a method different from fault pruning is introduced to avoid faults on RTL and gate-level models which are always masked by the system. This method focuses on determining which fault-injection scenarios to use based on the available list of faults. Compared to classic SFI approaches which inject HW faults randomly into randomly selected workloads, this approach maps which faults must be injected into each workload. Therefore, this method reduces the probability of faults being masked by the system.

Simulation Checkpointing

Checkpoint-restore methods [90], also known as snapshotting methods, have been used initially for HW-verification purposes. A HW model's states can be dumped to a file (checkpoint) during one simulation and restored from that saved checkpoint to run subsequent simulations (Fig. 1.2). As a result, simulation time is saved, and the overall verification effort of HW models is minimized. For instance, consider the verification effort of a microprocessor with a lengthy boot-up cycle. This phase is usually independent of the firmware code which is executed on the microprocessor. However, every time the microprocessor is simulated, the boot-up phase must also be executed. By saving a snapshot after the boot-up phase, new simulations can start from a more convenient point. This ability leads to faster simulation results.

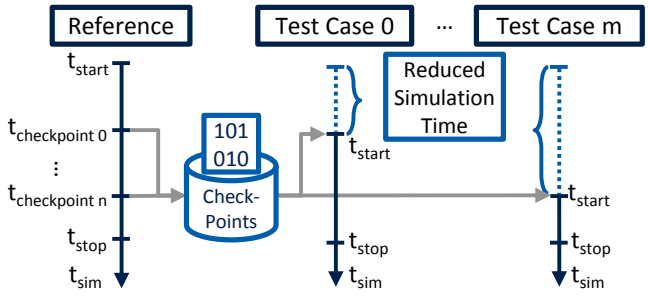


Figure 1.2.: Generic checkpoint-restore mechanism

In the context of safety verification, before faults are injected into a system, a simulation must first bring the system into the necessary state. Furthermore, various faults may be injected into the same system state which may then propagate and become failures. Checkpointing mechanisms can remove this unnecessary simulation time, thus speeding up fault-injection campaigns.

Checkpointing mechanisms can be grouped into three categories based on HW, software (SW), and virtual machine (VM) solutions.

HW-based checkpointing is a particular type of safety mechanism implemented in safety-critical SoCs [32], [91]. It relies on periodically saving a given system's state using individual checkpointing registers. The saved state is restored from the checkpoint in the event of error detection, thus, avoiding expensive system resets. However, this technique is useful during a SoC's operation, not for safety verification.

SW-based checkpointing is a widely used technique to create snapshots of HW models during a simulation [85]; it is supported by commercial HW simulators and has a standardized interface within the Verilog language via built-in methods (e.g., $save, $restart) [92]. Commercial simulators provide their implementation of the checkpointing mechanism which varies regarding hard-disk space requirements and stored information (e.g., HW models, type of HW simulator) [93]–[95]. Moreover, in the case the checkpointed HW models are changed (e.g., signals added or removed), snapshots can be reused only in limited conditions. This limitation considerably reduces their usefulness when debugging.

GDB can be successfully used to create checkpoints by dumping the whole memory content of a SW application or HW simulation and restoring it later [96]. However, GDB's checkpoints require substantially more hard-disk space than those of commercial tools; in this case, much more data are saved (e.g., cached data, memory buffers, data pointers) than just the HW model's states and simulator kernel. GDB's limitations have been addressed in [97] through the development of two new checkpointing mechanisms: an operating-system-level method and a process-level approach. However, these mechanisms create checkpoints which require several MBs of hard-disk space and multiple seconds to generate and store. Therefore, these techniques are still infeasible for fault-injection campaigns which need several thousand snapshots.

SW-based checkpointing is also applied to SystemC and TLM-based VPs [98]. In [99], particular checkpointing data types are used instead of SystemC signals and variables to save a VP's state during simulation. However, the approach is limited to SystemC methods only. Thus, SystemC threads are not checkpointed since they are operating-system dependent.

VM-based checkpointing of HW models uses a VM's built-in snapshotting mechanism to store a HW model's state [100]. This technique addresses the limitations of [99] because SystemC/TLM simulation-states can be implicitly saved as part of the VM's snapshot. However, this approach uses a considerable amount of hard-disk space because it saves the SystemC/TLM model's states

(i.e., including SystemC threads), and also all VM-specific information (e.g., caches, buffers, pointers).

In [86], [101], two checkpoint-restore methods have been integrated into different safety-verification flows to speed-up fault-injection simulations. Both methods create snapshots using commercial HW simulators for RTL and gate-level models. However, these methods suffer from four significant drawbacks. First, checkpoint generation requires a substantial amount of hard-disk space. Existing checkpointing tools save more than just the HW model's states (e.g., simulator kernel state, waveforms, scheduled events). Thus, particular systems have many signal states which must be saved but also a considerable amount of simulator information. Consequently, saving a sufficient number of snapshots for a fault-injection campaign risks exhausting the project's available hard-disk space. Second, checkpoints can only be generated during a running simulation. Thus, additional simulations must be run just to create new checkpoints. Third, checkpoint generation requires a considerable amount of time. The safety verification of large systems involves the creation of thousands of checkpoints. Fourth, snapshots are created manually, which results in sub-optimal fault-injection regressions.

1.5. Contributions of this Thesis

This thesis introduces several improvements to the field of safety verification of safety-critical SoCs in the automotive industry. This section outlines the contributions of this thesis to current state-of-the-art methods.

In this thesis, three fault-injection methods are introduced for the verification of VPs developed using SystemC and TLM. Fault-injection-based verification is a well-established practice in the aerospace, railway, computer, and other industries. With the adoption of ISO 26262, fault-injection verification has also become an integral part of the automotive industry. These fault-injection methods enable early safety-architecture exploration because they can be used during the early stages of SoC development, when safety mechanisms are evaluated for a safety-critical SoC. New VPs may immediately integrate the fault-injection features into their design. These methods also support non-intrusive fault injection. Thus, legacy models do not need to be updated. Additionally, faults are injected generically from the VP's test-bench. Therefore, VPs can be transferred to third parties (e.g., customers) without needing to disclose the VP's implementation details (i.e., protection of intellectual property).

Fault injection into VPs benefits from faster results because of more abstract HW models. However, this also creates a need for more abstract fault models. As shown in [9], the vast range of physical faults (e.g., electromigration, and neutron strikes) and transistor-level faults (e.g., bridging, open, and short circuits) can be successfully simplified at the gate level and RTL by injecting faults (e.g., bit-flips, stuck-ats) into memory cells, flip-flops, and other logic gates. However, at the abstract VP level, bit-flips and stuck-at faults are no longer useful. In this case, more abstract fault models are required, which are specialized on multi-bit components such as buses (e.g., bus fault), and even on higher-level components such as memories (e.g., instruction-memory fault, data-memory fault), register banks (e.g., register-read fault, register-write fault), and others. The three fault-injection methods introduced in this thesis support injection of classic single-bit faults (e.g., bit-flips, stuck-ats), but also modeling and injection of abstract faults.

With the increasing trend of performing safety verification on VPs, the question of result accuracy arises. It is essential to analyze the correlation of fault-injection results across multiple abstraction levels. In [102], when comparing results from VPs and gate-level fault injection, over 45% of the faults injected into VPs have led to false-positive results (i.e., failures which did not occur on the gate-level net-list). A similar correlation gap has been presented in [29] as well, where 25% of injected faults have led to false-positive results, when injecting faults into a CPU core. This thesis introduces two mechanisms which guarantee the correlation of fault-injection results on any SoC model and avoid false-positive results. One of the mechanisms, VERITAS, augments existing VPs with gate-level information from equivalent net-lists. The other mechanism, VERITAS++, transforms entire gate-level net-lists into abstract, optimized SystemC VPs.

To bridge the gap between safety analysis and safety verification, this thesis introduces a safety-verification flow, called SaVer. SaVer uses the fault-injection methods presented above to inject faults into VPs automatically. Faults are injected from a so-called fault library, which is a formalized database, which describes the fault-injection scenarios required to verify the SoC. This fault library is created automatically during safety analysis by mapping data, such as information about failure modes and safety mechanisms to modeled HW elements within the VP. After performing a fault-injection campaign, SaVer provides simulation reports, which can be used to prove the failure rates predicted by safety analysis.

Even on the VP level, safety-verification methods require a large number of simulations to verify the safety mechanisms of a SoC, and also to characterize

the system's fault-tolerance levels. In this thesis, several optimization methods are introduced to speed up fault-injection simulations, and reduce their number. One of these approaches analyzes the structure of gate-level net-lists, and reduces the number of redundant fault-injection locations. Another approach performs trace analysis of fault-free simulations using different simulation scenarios (i.e., SoC workloads). Based on this information, the approach determines the optimal number of faults which can be injected into the system. Furthermore, this method groups such faults based on each workload. Experimental results have shown an average fault reduction of over 95%. This thesis also introduces a checkpointing mechanism, which halves the duration of fault-injection campaigns. This speed-up factor has been previously impossible due to technological constraints. The checkpointing mechanism requires three orders of magnitude less hard-disk space than other state-of-the-art methods, and generates checkpoints one order of magnitude faster. Finally, SaVer allows multiple fault-injection simulations to run in parallel, a process which further speeds up the safety-verification process.

1.6. Previous Publications

The state-of-the-art methods and challenges of safety verification on VPs are presented in [103]. The fault-injection methods for VPs are given in [104]–[108]. The methods to improve the accuracy of fault injection into VPs are introduced in [109]–[113]. The method which analyzes simulation traces and optimizes fault-injection simulations on VPs is discussed in [114]. The checkpointing mechanism is introduced in [115]. The link between safety analysis and safety verification is documented in [116]–[122].

1.7. Structure of this Thesis

The remainder of this thesis is structured as follows. Chapter 2 presents the background of simulation-based fault injection and VP development. Chapter 3 introduces three fault-injection methods for VPs developed using SystemC, TLM, and C++. Chapter 4 describes the challenges of fault injection into VPs and introduces two methods to overcome them (i.e., VERITAS and VERITAS++). Chapter 5 introduces multiple optimization methods to speed-up the fault-injection process. Chapter 6 presents experimental results for several HW models, and quantifies the effectiveness of the contributions of this thesis. Chapter 7 concludes the thesis.

2. Safety-Verification of Hardware Models

This chapter presents the basics of virtual prototyping using SystemC and TLM and the basics of the fault-injection methodology.

2.1. Virtual Prototypes

The SystemC and TLM libraries have become the *de facto* standard for virtual prototyping [98], [123]–[125]. The IEEE 1666-2011 standard defines SystemC and TLM as "an ANSI standard C++ class library for system and HW design for use by designers and architects who need to address complex systems that are a hybrid between HW and SW" [98]. System architects, also called concept engineers, typically use SystemC and TLM to model large-scale systems at an abstract SW-accessible register-accurate level. This modeling method allows HW developers to perform *early architecture exploration* and SW developers to design and validate their applications early in the development phase of a system. Moreover, safety-verification methods are being developed for SystemC/TLM-based VPs to perform *early safety-architecture exploration*.

With the steady increase in complexity of HW models, current gate-level net-lists and RTL models suffer from long development and verification times. Consequently, complex SoCs may fail to meet the market's growing constraints. For this reason, researchers are developing and standardizing leaner models such as VPs based on SystemC and TLM [98]. These VPs aim to speed up HW simulations, shorten development and debugging iterations, and improve the initial design of HW systems. Thus, VPs are a robust modeling approach complementary to traditional RTL and gate-level modeling.

Besides their quick turn around, SystemC VPs support binding to high-level programming languages (e.g., C++, Python). Such VPs benefit from interoperability with standard and third-party libraries supported by these languages. Compared to RTL design, VPs benefit from object-oriented programming pat-

terns which improve their re-usability and programming flexibility [98], [106]. Furthermore, HW designers can use VPs as reference implementations to enable parallel system development. Hence, VPs allow concurrent design and verification of firmware and HW models. Consequently, firmware developers can directly execute their code on a VP with predefined accuracy (i.e., clock-cycle accuracy, approximately timed, loosely timed, or untimed) instead of waiting to receive a running HW module or emulator. However, VPs are not suitable for SoC sign-off as they are too abstract.

2.1.1. SystemC

Similar to RTL and gate-level modeling, SystemC-based virtual prototyping encapsulates HW components into modules, which can be nested within other modules to create a system hierarchy (Fig. 2.1). Two or more modules can be connected using ports (e.g., input or output) which describe the directional flow of data. For instance, data from the output port of one module flow into the input port of another module. Additionally, data from input ports of a module flow into the input ports of sub-modules. Since ports do not store the information passed between modules, ports must be connected via signals. The SystemC library contains a built-in event-driven simulator which can compile and execute SystemC models.

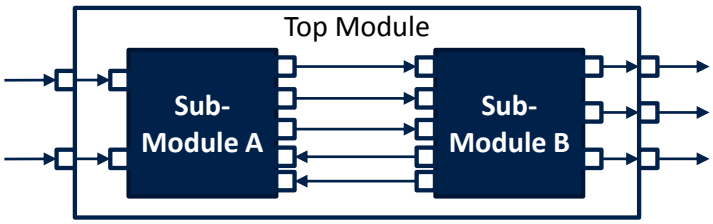


Figure 2.1.: Simple SystemC model

2.1.2. TLM

Because SystemC models follow a similar modeling paradigm to RTL and gate-level models, the TLM library has been created to add another layer of abstraction to virtual prototyping. TLM models structure HW information hierarchically using so-called initiator and target modules (Fig. 2.2). The communication among these modules is achieved using initiator and target sockets instead of ports. Sockets pass data around modules using abstract payloads instead of

bit-wise signals. Furthermore, payloads are transported using SW calls instead of scheduled events. Thus, TLM models further improve the execution speed of VPs. TLM communication is modeled as follows. The initiator model (e.g., a CPU core) initiates a SW call to the instruction memory to fetch an instruction. In doing so, it passes a payload which contains the command for the next instruction. The target model (e.g., the instruction memory) implements the SW method, which the initiator has called. This SW method stores the next instruction in the payload, increments the program counter (i.e., an internal variable), and returns a payload with the instruction. Then, the initiator sends the instruction to the instruction decoder block, and the process repeats.

Figure 2.2.: Basic TLM model

SystemC and TLM models can be linked together using a wrapper module (Fig. 2.3). This module synchronizes the communication across the two models by packing pin-level information into payloads, or unpacking payloads and providing information on individual pins.

Figure 2.3.: TLM-SystemC wrapper module which synchronizes the communication across SystemC and TLM models

TLM models benefit from the following modeling styles [98]:

Un-timed do not simulate any timing information. This modeling style is similar to behavioral HW models executed on instruction-set simulators such as QEMU [126]. Such models utilize the blocking-transport method. The communication across modules is simple and uses dedicated payloads. In other words, payloads are not shared across multiple modules.

Loosely-timed contain limited timing information passed during payload transactions across several modules. Similar to un-timed models, this style

utilizes the blocking-transport method and has a simple communication scheme across modules.

Approximately-timed have complex interdependencies, and their communication is modeled using several timing phases. Compared to the other two modeling styles, payloads are shared and passed across multiple modules.

2.2. Simulation-Based Fault Injection

Before its adoption in the automotive industry via ISO 26262, fault injection has already been utilized in several other industries such as avionics, computing, and networking. This section explores the theory of simulation-based fault injection and its components.

2.2.1. Fault-Injection Environment

The first generic simulation-based fault-injection framework was introduced in 1997, and it was based on the structure of a test-bench [41]. A test-bench is a closed system used to test the functionality of a HW model, which mimics the operational environment of a system under verification up to a predefined level of detail. The traditional test-bench framework contains a stimulus generator, monitor for output data from the system, a simulation controller, and a data analyzer (Fig. 2.4). The stimulus generator drives data either randomly or targeted (i.e., using a stimulus library) onto the system's inputs. Stimuli can be provided statically (e.g., test sequences written in files) [127], symbolically (i.e., symbolic simulation) [128], or randomly [129]. The monitor detects changes at the model's output ports and sends the observed data to an analyzer for processing. The data analyzer determines whether the system has executed its task correctly (based on the provided input stimuli). Finally, the controller regulates the driven stimulus and the lifetime of a simulation.

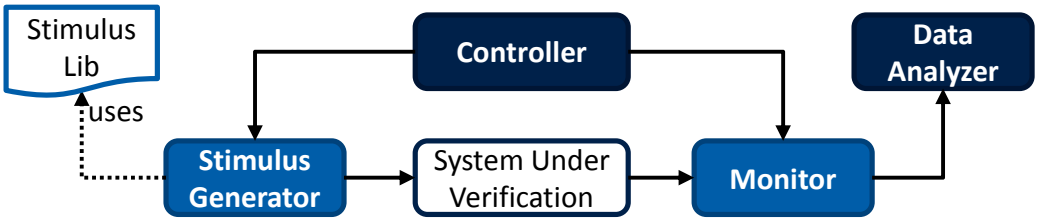


Figure 2.4.: Generic test-bench framework

The fault-injection framework introduced in [41] extends the traditional testbench's structure by adding two new components: a fault injector and a fault library (Fig. 2.5). The fault injector works similarly to a stimulus generator, but instead of driving stimuli, it inserts abstract representations of physical HW faults into the HW system. This information is stored in the fault library. Finally, the fault-injection framework monitors the system's outputs and internal states to determine the injected fault's propagation path. After fault injection, results are logged, and coverage information is computed referring to the injected fault's effect (e.g., detected, masked, not detected, latent) and propagation path. Next, the system's vulnerability is calculated and continuously updated after each new fault injection until all simulations are executed.

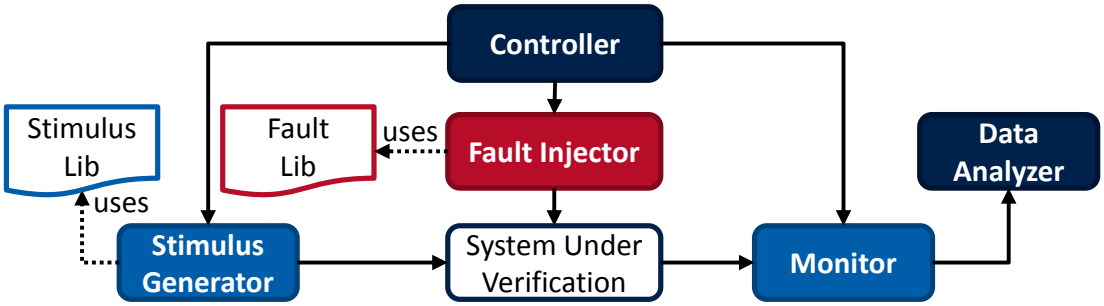


Figure 2.5.: Generic fault-injection testbench [41]

Fault-injection-based safety verification requires bug-free HW systems. The presence of development faults (i.e., bugs) may lead to inaccurate fault-injection results. For instance, a bug may hinder fault propagation or may lead to incorrect fault propagation. Consequently, it is difficult if not even impossible to determine whether the observed fault effect has been caused (or masked) by the bug. Thus, fault injection into safety-critical systems requires comprehensive functional verification beforehand.

2.2.2. Fault-Injection Attributes

Fault-injection frameworks require fault libraries which hold information vital for fault-injection simulations. Generic fault-injection methods need only three essential attributes to successfully inject faults into a HW model during simulation [87], [130], [131]: (i) fault-injection location, (ii) fault-activation time, and (iii) fault type. Even though different fault-injection tools may also use other fault-injection attributes, these three are fundamental and are always present (Fig. 2.6).

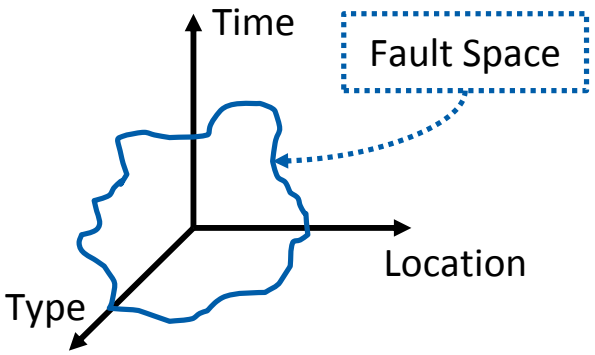


Figure 2.6.: Fault verification space [89]

Fault-injection locations represent internal elements within a HW system model (e.g., signals, registers, latches, memory cells, variables) into which faults can be injected. Unlike system ports, fault-injection locations are not accessible from outside of the system by the stimulus generator. Fault-activation times are the instance of execution time used to inject a specific fault into a fault-injection location (e.g., at 5 ns, 10 ms). Fault types are the fault models used to inject faults into a system at a specific abstraction level (e.g., VP, RTL, gate-level). These fault models are abstract representations of actual HW faults (e.g., electromigration, aging effects, neutron strikes).

2.2.3. Fault-Verification Space

The fault-verification space of safety-critical HW systems is the cross-product of the attributes mentioned above (Fig. 2.6). When describing a system's fault-verification space, fault-injection locations (e.g., registers) and fault types (i.e., permanent or transient) are extracted from the system's structure. However, fault-activation times, which cause system failures, are usually determined statistically through simulation [89], [102]. In other words, faults are injected at random simulation times, and their effects are observed at the end of the simulation. This difficulty is caused by the combination of the system's structure (including safety mechanisms), operational profile (i.e., test workloads), and mode of operation (e.g., *sleep mode*, *test mode*). Similarly, it is difficult to determine whether a system is inherently safe against a particular fault. In other words, can a distinct fault be injected into a specific system location and never lead to a failure for any fault-activation time and system configuration? If this question cannot be quickly answered analytically, statistical verification is necessary across a sufficient number of test workloads and fault-activation times.

Because of the inherent complexity of determining correct fault-activation times, researchers such as [89], [102] consider the fault-verification space to be infinitely large. However, it is not necessary to test all fault-injection locations and fault types across infinitely many fault-activation times. Instead, given a system's structure and operational profile, it is sufficient to define test workloads, for which an activated fault causes a system failure. If no workload can be defined for such fault-activation times, the analyzed SoC can be considered safe from this fault effect, and the fault can be removed from the fault-injection campaign. Consequently, only a finite number of fault-activation times is necessary to cause all possible failures of a system. Furthermore, the number of fault-injection locations and fault types within a system are also finite. Hence, the fault-verification space, even though large, is finite in size.

2.2.4. Fault-Propagation Paths

Faults can propagate through a system on the same abstraction level, and also at different abstraction levels. The authors of [132] have documented two types of fault propagation: horizontal and vertical.

Horizontal fault propagation occurs when a fault occurs in one system (e.g., System A), and propagates to another system (e.g., System B) (Fig. 2.7). In this case, two types of fault-propagation paths become apparent: internal and external. **Internal propagation** occurs when activated faults propagate through the first system [83]. This propagation either leads to faults being masked by the system or to faults reaching the system output and becoming failures. **External propagation** occurs when failures from the first system propagate to the next system and cause faults.

Vertical fault propagation occurs when a fault defined on one abstraction level (e.g., RTL model) represents a failure on a lower abstraction level (e.g., gate-level net-list) (Fig. 2.8). Conversely, fault effects, also called errors, on detailed abstraction levels (e.g., gate-level net-lists) are considered faults on higher abstraction levels (e.g., RTL models or VPs).

2.2.5. Types of Hardware Simulators

Fault simulation has evolved from static analysis of memory cells, logic elements, and flip-flops to analysis of clock trees, transient faults, and fault-masking

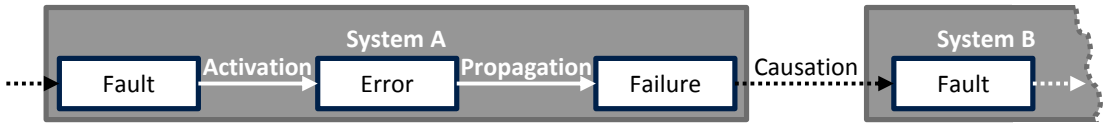


Figure 2.7.: Horizontal fault-error-failure propagation chain [83]

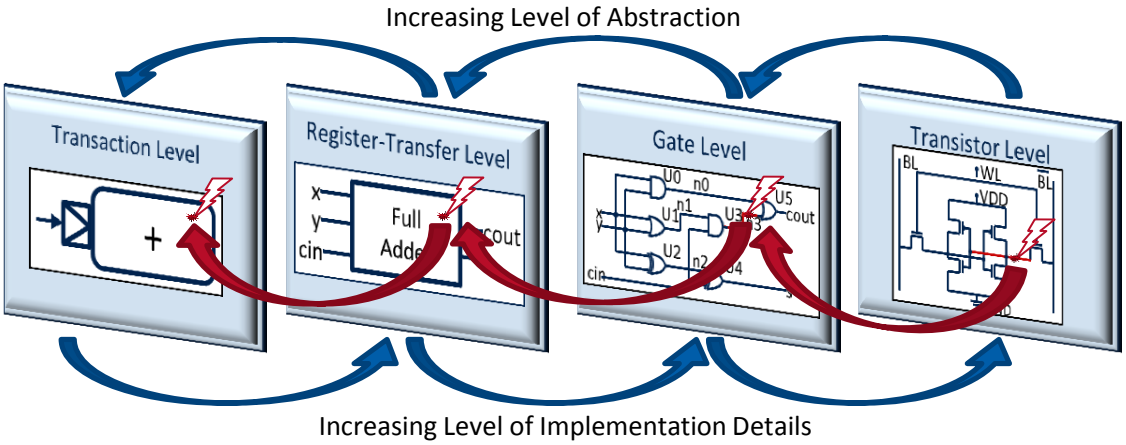


Figure 2.8.: Vertical fault-error-failure propagation chain

effects [29]. As already discussed previously, simulation-based fault-injection methods use fault libraries and verification environments to emulate fault effects on the model of a HW system. Such methods utilize specialized HW simulators to execute the functionality (behavior) of a HW system on one or more PCs. HW models are developed using specific HDLs (e.g., Verilog, VHDL), SystemC, or TLM. Furthermore, the faults injected into these HW models are abstract representations of real faults.

Fault simulation requires a HW simulator which mimics the system model's functionality. Many types of HW simulators are available for simulation-based safety verification, such as cycle-accurate simulators (CASs), event-driven simulators (EDSs), fault simulators, host-compiled simulators, and others [36]. These simulators have different advantages based on what type of HW models they support. For this reason, an adequate simulator must be chosen based on the model's abstraction level and the verification intent (e.g., performance testing, safety verification).

CASs execute the complete HW model at every clock cycle [133], [134]. They have high performance for single-core systems, but the execution degrades significantly on multi-core systems. **EDSs** only simulate active components of a

HW model on each clock cycle [93], [95]. Active components are determined using cascading events. In other words, active components trigger the next set of active components. These events are prioritized and ordered by a scheduler and stored in event queues. The performance of EDSs depends on the size and complexity of the simulated system. Even though EDSs tend to have a lower simulation performance on single-core systems, their performance considerably increases on multi-core systems [36]. **Fault simulators** execute multiple fault-injection simulations in parallel using the same simulator kernel [135]. By comparison, CASs and EDSs can also run in parallel, but they use a separate kernel for each simulation. By sharing the same simulator kernel across multiple fault-injection simulations, fault simulators use less memory and require less computational power than CASs and EDSs. The main drawback of fault simulators is *hyperactivity*, which appears when fault injection leads to long and unique fault-propagation paths [136]. Such scenarios considerably decrease the overall simulation performance. **Host-compiled simulators**, also known as **compiled-code simulators**, compile the system model into machine code and execute it on a target machine (e.g., Intel X86_64 architecture) [133], [134]. These simulators are on average twice as fast as EDSs [36].

2.2.6. Fault Models

As already discussed, fault-injection methods utilize abstract representations of random HW faults, also called fault models, during fault-injection experiments. Fault models emulate the effects of real HW faults. Their structure depends on the type of fault-injection method (e.g., physical, emulation, or simulation testing). In simulation-based safety verification, fault models also depend on the HW model's abstraction level. Thus, it is important to distinguish among gate level, RTL, and VP fault models. This section explains the causes of random HW faults, describes the most commonly used fault models, and classifies them based on the HW abstraction level where they are used (Table 2.1).

Types and Causes of Hardware Faults

Faults which affect modern semiconductor systems can be modeled as the effect of natural radiation, mechanical stress (e.g., wire breaks), aging (e.g., electromigration), and others. On the one hand, ionizing doses accumulated by electrons and protons may permanently damage a transistor's switching capabilities [29]. On the other hand, neutrons, protons, and heavy ions may lead

Table 2.1.: Example of fault-model classifications based on abstraction level

Abstraction Level	Permanent	Transient
Transistor Level	Single-event gate ruptures Single-event burnouts	Single-event latch-ups Electric latch-ups
Circuit Level (RTL Model or Gate-Level Net-List)	Stuck-at-0 Stuck-at-1 Bridging Timing (transition delay)	Single-event upsets Single-event transients Single-event functional interrupts Spatial/temporal multi-bit upsets
Virtual Prototype (SystemC or TLM Model)	Register read Register write Bus access CPU pipeline stage	

to reversible bit-flips in combinational and sequential gates, called single-event effects [29]. In this case, each particle can cause upsets in the affected components without the need to accumulate ionizing doses. The most critical single-event effects on modern semiconductor devices are permanent and transient faults.

Permanent faults, also called hard errors, are mainly caused by physical faults which are either the result of manufacturing variations or occur over time in the SoC due to random dopant fluctuation [137], electromigration [13], and other aging effects [13], [138]–[140]. These *hard* effects are essential for a range of applications: medical, military, space, security, automotive, and others.

Transient faults, also called soft errors, are mainly the result of radiation effects which affect the electric charge within logic gates and memory elements. These create bit-flips in the digital system [9]. Contrary to hard errors, soft errors do not cause permanent damage to the affected circuit, and thus can be eliminated by writing new data to the affected system. Within the terrestrial radiation environment, soft errors are induced by alpha particles from the SoC's package [10], high-energy cosmic radiation (i.e., galactic radiation) [10], [141], and low-energy cosmic radiation (i.e., neutron particles) [10].

Transistor-Level Fault Models

Transistor-level fault models describe effects which are either common to all transistors or particular to some. **Single-event gate ruptures** are permanent fault effects which lead to breakdowns in the gates of transistors. At the physical

level, these faults create transient electric fields across the gate oxide of power metal-oxide-semiconductor-field-effect-transistors (MOSFETs) [12]. **Single-event burnouts** affect power MOSFETs and create a forward bias in the parasitic transistor [12]. **Single-event latch-ups** are caused when the complementary metal-oxide-semiconductor (CMOS) parasitic bipolar transistors between well and substrate are turned on by radiation effects. These faults can only be removed after powering down the system and can lead to permanent damage to the affected SoC [10], [29]. **Electric latch-ups** are the same as single-event latch-ups but are caused by over-voltage. They also do not require the system to be powered down to remove their effect [10].

Circuit-Level Fault Models

Individual transistor faults are abstracted at the circuit level, and modeled based on the behavior of logic gates [9]. Since logic gates are composed of different transistors, fault effects at the gate abstraction level may be the result of any number of faulted transistors. Thus, gate-level fault models mainly describe a delayed output or an output whose value differs from an expected logical value.

Stuck-at faults create permanent effects which fix a HW signal (i.e., metal wire) to a specific logic value: '1' (i.e., stuck-at-1) or '0' (i.e., stuck-at-0) [142]. **Bridging and timing faults** are well-known fault effects discovered for several decades which are mainly handled through ATPG testing [143], [144]. Bridging faults represent short circuits between two or more signals of a HW system [142]. Such faults usually occur between signals which are near each other. Thus, bridging faults can only be effectively modeled in the presence of gate-level layout information. Timing faults, also called transient-delay faults, represent the effects of faults which lead to a delayed response in the system. Here, the affected system parts calculate the correct result, but the response is observed later than in the fault-free scenario. Timing faults are used to model a series of physical faults caused by temperature, supply noise, process variations, and others [142]. **Single-event upsets** represent a reversible bit inversion on a memory cell, latch, or flip-flop during execution [10]. **Single-event transients** create a temporary change in a combinational gate's output voltage. They become single-event upsets after the bit-flip gets propagated to a sequential logic element and becomes latched there [10].

Virtual-Prototype Fault Models

At the VP abstraction level, fault models become even more abstract and represent combinations of one or more random HW faults. Here, transistor and circuit-level faults (e.g., stuck-at faults, timing faults) are too specific and require considerable modeling effort. In this case, faults models are defined based on system components and sub-components (e.g., bus-access fault, cache fault, register-read fault, register-write fault). Thus, the primary benefit of VP fault models is their reduced granularity. Furthermore, since VP fault models are tailored for a specific part of a system, they provide a human-readable description of the analyzed fault effects. For instance, a *bus-access fault* models a fault effect which occurs when accessing a bus. Additionally, a system without a bus model requires no bus-fault injection.

VP fault models can be broken down into more granular faults (e.g., bus read-access fault, bus-arbitration fault). Theoretically, this process may be repeated until the granularity level reaches that of the original circuit-level faults. However, this approach loses all the benefits provided by the abstraction in the first place: early availability and reduced modeling effort. However, since faults are specific to a system's component, they must be manually defined for every component within the system. As a result, it is not possible to reuse a bus fault as a CPU-core fault or as a cache fault. In contrast, generic fault models (e.g., stuck-at, bit-flip) from the circuit level can be applied automatically to any signal or gate within a system.

2.3. Summary

Fault simulation can be applied to system models developed on different levels of abstraction such as VPs, RTL models, and gate-level net-lists. During fault simulation, a verification environment stimulates system inputs with real-life data and monitors system outputs for fault effects. Additionally, a fault-injection mechanism inserts faults into internal system states either during a simulation or before the start of a simulation. This mechanism uses fault models stored inside a fault library to simulate the effects of physical faults on the system models. The outcomes of injected faults are determined by automatically comparing the simulation results to those from a fault-free simulation also called a reference simulation. The VPs, RTL models, and gate-level net-lists are executed using specialized HW simulators. The execution speed of the HW model can be influenced by the type of the simulator (e.g., CAS, EDS, compiled-code simulators).

3. Generic Fault-Injection Methods for Virtual Prototypes

3.1. Introduction

As already discussed, existing fault-injection methodologies for SystemC/TLM-based VPs introduce a significant simulation overhead to the analyzed models, require complicated model-dependent adapters to inject faults, and need modifications of the original model. This chapter introduces several fault-injection methods developed to tackle these drawbacks.

3.2. SCFIT

SCFIT (SystemC Fault-Injection Tool) is a fault-injection tool for SystemC and TLM models [105]. SCFIT uses GDB to control the SystemC simulation kernel and to gain access to a compiled SystemC model's data types at runtime (Fig. 3.1). Faults are injected during simulation using breakpoints and watchpoints set on:

- SystemC ports, signals, and processes
- TLM sockets and payloads
- C++ variables

These breakpoints and watchpoints are set and controlled using Python scripts connected to GDB through GDB's Python API [96]. This mechanism allows direct access to public, private, and protected data types during a SystemC simulation without changing the original model.

As the name suggests, breakpoints and watchpoints stop the execution of a SystemC/TLM model when a specific SystemC process is executed, or a variable's value is updated. Standard Intel x86_64 processors offer up to four dedicated HW breakpoints and watchpoints [145], [146]. In other words, multiple simultaneous faults may be injected into a running SystemC model. If more

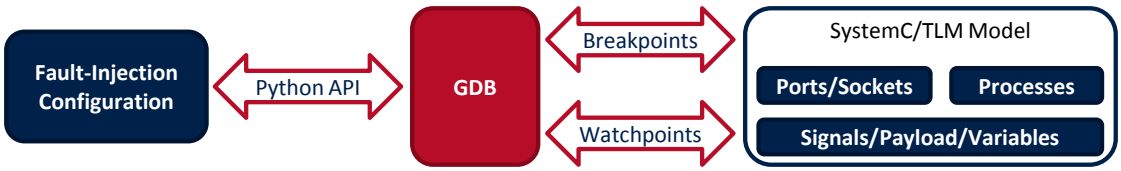


Figure 3.1.: SCFIT's fault-injection mechanism

than four HW breakpoints and watchpoints are required, GDB automatically creates SW ones. However, it is important to avoid software breakpoints and watchpoints because they lack significant computational resources and slow down simulation performance.

3.2.1. Fault-Injection Locations

Faults are injected by first setting one or more breakpoints on the desired fault-injection location (Table 3.1). SCFIT is designed to insert faults into SystemC signals, SystemC ports, TLM sockets, TLM payloads, static, and non-static C++ variables. These data types are used extensively in SystemC/TLM-based HW modeling to describe transactions through-out a HW system. Other SystemC/TLM-specific data types such as events, processes, or the quantum keeper are not suitable fault-injection locations because they are mainly associated with the system's structure and with the simulation kernel of a SystemC/TLM-based VP. Nevertheless, SCFIT can be easily adapted to support fault injection into any SystemC/TLM data type.

Table 3.1.: SCFIT's requirements for placing breakpoints and watchpoints

Data Type	Location of Breakpoints/Watchpoints
SystemC Signal	• SystemC process or C++ method which accesses the signal • Write method and overloaded operator= method or • Read method and overloaded cast operator
TLM Socket	• b_transport method • nb_transport_fw method • nb_transport_bw method
TLM Payload	• Set and get methods of each payload attribute (e.g., address, data, command)
C++ Static Variable	• SystemC process or C++ method which accesses the variable • Watchpoint on the variable itself
C++ Non-Static Variable	• SystemC process or C++ method where the variable is declared • Watchpoint on the variable itself

Each SystemC signal has a read and write method, which can be called during simulation to transfer data from one signal to the next (e.g., o.write(i.read())). Additionally, since SystemC signals are C++ classes, read and write methods may be replaced by the class's overloaded methods (e.g., o = i). Faults may be injected either when a signal is read or when it is written. Thus, a breakpoint is required on each of read or write methods (two in total). A breakpoint is also needed on the SystemC process or C++ method which accesses the signal. Therefore, a total of three HW breakpoints are necessary for any fault injection into SystemC signals.

TLM sockets have three transport methods: blocking, non-blocking forward, and non-blocking backward. Hence, a total of three breakpoints are required (i.e., one on each transport method) to gain access to the TLM socket's payload and time-delay attribute. Alternatively, the TLM payload may be accessed by setting a breakpoint on its setter or getter methods for each attribute (e.g., address, data, command). Thus, each TLM-payload attribute requires a pair of breakpoints (i.e., one on the setter method and the other on the getter method).

GDB views static C++ variables within the program's global scope and therefore, only one HW watchpoint is necessary to access them. If it is essential to determine which SystemC process or C++ method accessed the static variable, a breakpoint may also be placed on that process or method.

Non-static C++ variables are only visible within the class or method where they are declared. Thus, a breakpoint is required on the C++ method or SystemC process which updates or reads the variable. Then, a watchpoint must be set on the variable to stop the simulator's execution when the variable is accessed. After the simulation exits the method's scope, the watchpoint is deleted and must be created again when the simulator reenters the method's scope.

Before setting breakpoints to SystemC and TLM template-based data types, it is important to determine the argument type of each template. From the perspective of C++, a Boolean SystemC signal (i.e., sc_signal<bool>) is different from an integer SystemC signal (i.e., sc_signal<int>). C++ also considers the methods of classes with different template arguments to have different scopes and signatures. Consequently, even though both signals have read and write methods and are of type sc_signal, breakpoints must be set for each SystemC data type separately. The same reasoning applies to TLM sockets whose complexity increases as TLM sockets use multiple template arguments.

During a simulation, breakpoints are triggered for all possible instances of a specific data type. However, fault injection occurs only into a specific instance of a SystemC, TLM, or C++ data type. Thus, SCFIT uses SystemC's introspection

to determine the correct instance where to inject faults. This is achieved by monitoring the name attribute of a TLM socket, SystemC process, or signal. However, TLM payloads do not have such an attribute. In this case, the most flexible method of accessing a TLM payload's attributes is via a TLM socket. C++ variables also lack a name attribute. Nevertheless, watchpoints are sufficient to access the correct variable declaration.

3.2.2. Fault Models

Two types of faults can be injected using SCFIT: transient (i.e., bit-flips) and permanent faults (i.e., stuck-at faults). Transient faults are injected at a user-defined simulation time. Their effects may be overwritten by the simulator at the next access of the signal. Thus, the breakpoints are programmed to be triggered at a user-defined simulation time and for a specific instance of a data type. After injecting the fault (i.e., modifying the object's value), breakpoints and watchpoints are deleted, and the simulation runs until completion. Permanent faults are injected every time a breakpoint is triggered since a SystemC/TLM simulation can override the injected fault. Their effect starts at the beginning of the simulation and persists throughout the whole simulation. Thus, breakpoints and watchpoints are active during the entire simulation and continuously set the data type's value (i.e., stuck-at-0 or stuck-at-1) each time they are triggered.

3.2.3. Fault-Injection Flow

SCFIT uses the same test-bench structure as in Fig. 2.5. The system's original SystemC/TLM test-bench is used to provide real stimuli. SCFIT injects faults and controls the SystemC simulation kernel using GDB and Python scripts. Faults are stored in a text-based file, called a fault library. SCFIT parses the fault library during the configuration phase, and injects faults into the SystemC VP. Faults are described based on their three main attributes: location (i.e., SystemC, TLM, or C++ data-type instance), type (i.e., permanent or transient), and fault-injection time.

SCFIT's fault-injection flow has four phases (Fig. 3.2). First, the SystemC models and the test-bench are compiled. Additionally, SCFIT is configured to set breakpoints and watchpoints on specific data-type instances (e.g., signals, payloads, variables) derived from the fault library. Next, the SystemC models are loaded into GDB and SCFIT is initialized by setting breakpoints. During the simulation, breakpoints and watchpoints are triggered automatically, and

faults are injected. Results are gathered by the SystemC/TLM test-bench in the form of a value-change dump (VCD) file. Finally, SCFIT analyzes fault-injection simulation outcomes by comparing the results of a fault-free simulation, also known as a reference simulation, to those of each fault-injection simulation (Fig. 3.3). In the case of a mismatch between the two simulations, the injected fault caused a failure. Otherwise, the fault is considered to have been masked by the system.

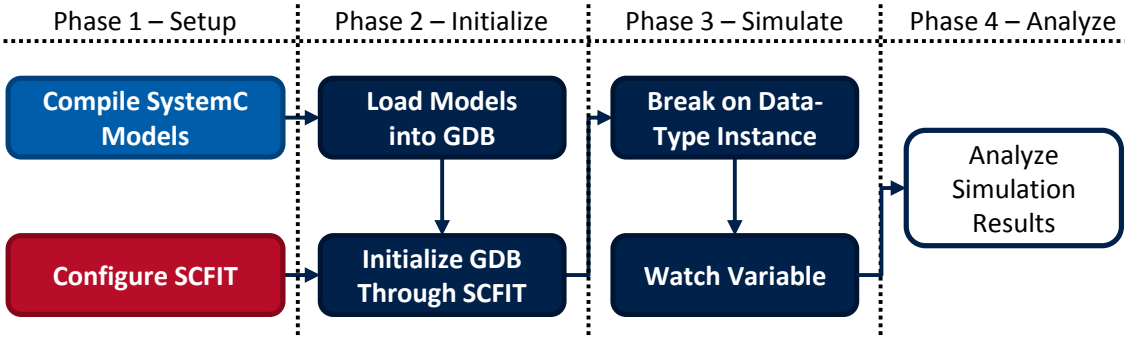


Figure 3.2.: SCFIT's fault-injection execution flow

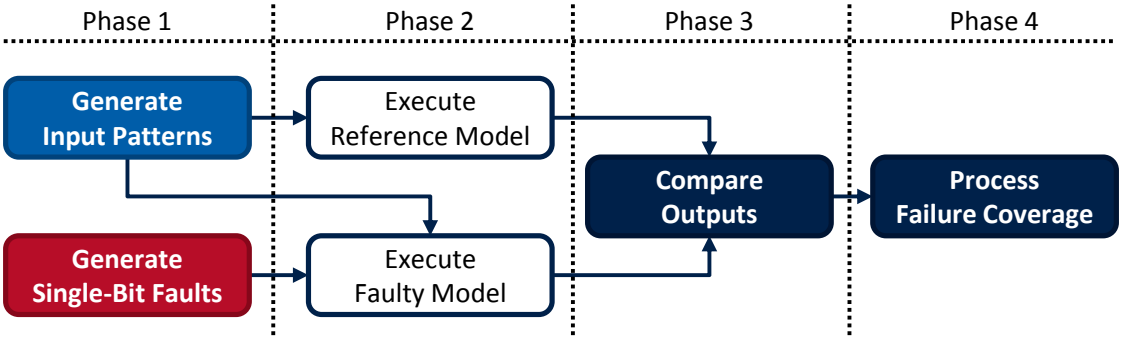


Figure 3.3.: SCFIT's fault-injection flow

SCFIT can inject any number of faults into SystemC designs. However, due to the HW limitation of current processors, a maximum of one fault per simulation is recommended to maintain minimal computational overhead (see Chapter 6.5.1). Chapter 3.3 presents a solution to overcome this limitation.

3.2.4. Model-Based Automation and Graphical User Interface

SCFIT benefits from a model-based graphical user interface (GUI) which generates a fault library and documentation about fault-injection campaigns based on

user inputs [104]. The GUI parses the SystemC hierarchy (e.g., modules, ports, signals) provided either by the SystemC library or by an external application. The information is stored into a data model which is represented graphically. Users can map corresponding locations (e.g., signals, payloads) to fault types and set an injection time for transient faults. In the example from Fig. 3.4, a multi-bit fault with value 42 is injected into output port *d_out1* at simulation time 600 ns.

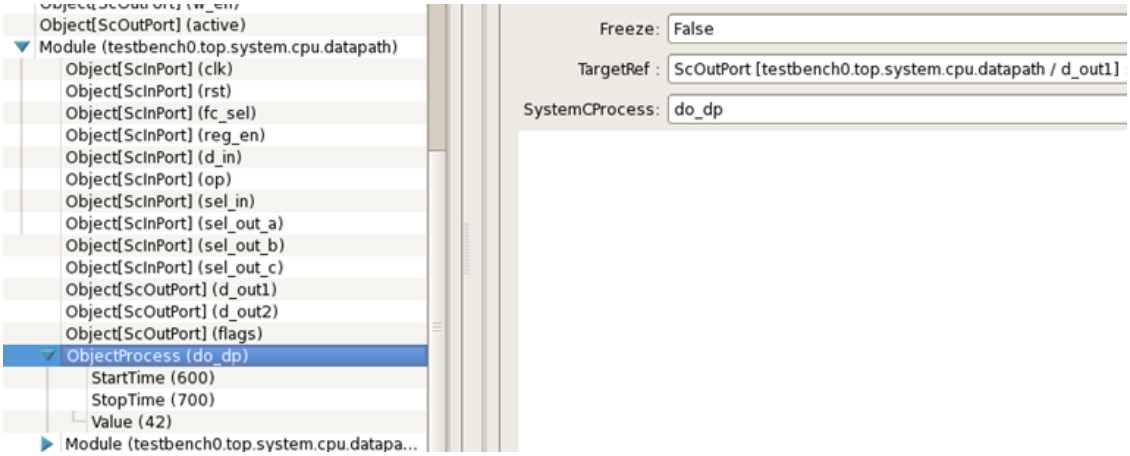


Figure 3.4.: Example of SCFIT's GUI

The generated fault library is a series of Python scripts with all the information required to run a fault-injection campaign. The generated documentation is a text file which describes scenarios performed during the fault-injection campaign. The example presented in Fig. 3.5 shows four fault-injection simulations. Here, different faults are injected into three C++ variables (e.g., value 3 in variable top0.fsm0.y at simulation time 40 ns).

3.3. Simulator Commands for SystemC

Safety verification may be further improved by replacing the GDB-based approach with a method based on simulator commands. Simulator commands represent actions which can be programmed on a HW simulator, usually via a dedicated API, to control the behavior of a simulation. Each simulator implements its list of simulation commands. The most widely available ones are commands for probing or modifying a signal's value during simulation. Simulator commands are non-intrusive because the HW model's source code

Fault-Injection Documentation for SystemC Test-Case FSM_PORT.
Automatically Generated with SCFIT Gen.

Manipulated Variables:
top0.fsm0.y
top0.fsm0.a
top0.fsm1.y

Fault-Injection Scenarios:
@40 SC_NS until 80 SC_NS: force **write**-fault in **top0.fsm0.y** with value **3**.
@90 SC_NS until 160 SC_NS: force **write**-fault in **top0.fsm0.y** with value ranging **from 15 to 30**.
@90 SC_NS until 130 SC_NS: force **write**-fault in **top0.fsm1.y** with value **70**.
@80 SC_MS until 90 SC_SEC: force **read**-fault in **top0.fsm0.a** with value **0**.

Figure 3.5.: Example of SCFIT's generated fault-injection report

does not require modifications. Additionally, they are a more efficient fault-injection method compared to saboteurs and code mutation [48]. They are generic, re-usable on any HW model, and have low simulation overhead which is advantageous when running thousands of fault-injection simulations.

The current implementation of the SystemC simulator (i.e., SystemC 2.3.1) does not support simulator commands. For this reason, this thesis introduces new data types to the SystemC library, which support fault-injection methods (e.g., fault-injection signal, buffer, variable). The newly added fault-injection objects (FIOs) use object-oriented inheritance to retain all properties and attributes of the original SystemC data types (e.g., SystemC signal, buffer) while adding fault-injection methods for transient (e.g., bit flip) and permanent faults (e.g., stuck-at-0, stuck-at-1). Similar to SystemC data types, FIOs also use C++ templates to make the new data types generic and reusable. This implementation is particularly useful for injecting faults into C++ variables. In this case, standard Boolean, integer, float, etc. variables can be replaced by a generic FIO specialized for the type of the specific variable.

FIOs are extensions to SystemC's read and write accesses as described in the previous section. The additional fault-injection methods contained by FIOs are used as software switches to turn on a specific fault effect. These methods allow SystemC components to emulate the occurrence of HW faults during simulation. Besides this, fault-free simulations can be run merely by not calling any fault-injection method from the test-bench. This mechanism allows fault-injection regressions to be executed on the same compiled SystemC model without needing to change the model for a new fault-injection simulation.

Thanks to their implementation, test-benches can access FIOs declared as

private or protected within a SystemC model. Upon instantiation, FIOs are automatically registered to a FIO manager which can be accessed from anywhere within the test-bench (Fig. 3.6). The FIO manager maps each FIO instance to its SystemC hierarchical name. After obtaining the desired FIO, faults may be injected at any time during the simulation by calling the appropriate fault-injection method.

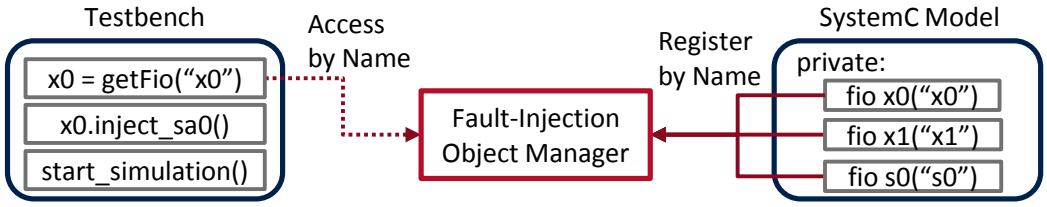


Figure 3.6.: Registration of fault-injection objects to fault-injection-object manager and access from testbench

Simulator commands successfully overcome the limitations of SCFIT. First, SystemC models can be sent to customers or third parties without the risk of divulging protected intellectual property. This feature is made possible because simulator commands do not require the SystemC models to be compiled in debug mode (i.e., using GCC's '-g' option). In turn, the SystemC model's implementation remains hidden from the customer. Second, this approach does not use any breakpoints or watchpoints. Since FIOs are not bound by the four-HW-breakpoints limitation, any number of faults can be injected into SystemC models with almost negligible simulation overhead. Third, FIOs have smaller simulation overhead compared to SCFIT. The performance measurements of FIOs are described in Chapter 6.5.3.

3.4. Simulator Commands for SystemC/TLM

Similar to the SystemC library, the TLM extension to SystemC does not support simulator commands. Additionally, the TLM library's implementation makes it impossible to use FIOs as described in the previous section. Two main reasons cause this limitation. First, TLM models are more abstract than SystemC models, which renders TLM fault models more abstract, too. Second, when comparing data types, TLM sockets and payloads have more complex data-transfer algorithms compared to the pure read and write access patterns of RTL and SystemC signals. Consequently, this thesis presents three TLM extensions which introduce simulator commands to TLM models: injectable interface,

injectable payload, and injectable sockets. These extensions are described in the following sections.

3.4.1. Injectable Interface

The injectable interface is an extension of the original TLM backward and forward interfaces (Fig. 3.7). Besides the typical virtual functions defining the blocking and non-blocking interfaces, a configuration data member (i.e., fault_mode) is used to activate fault injection during simulation. The data member is configurable to set the interface in normal or in fault-injection mode. In normal mode, the system continues its ordinary functions calling TLM interface routines. Once the simulation goes into fault-injection mode, the tlm_fault_inject transport method is called to inject faults into the interface model by corrupting the generic TLM payload's attributes. This injectable interface is usable in the early system-development phase, in which the communication protocol between two or more TLM modules is not yet fully defined and implemented. Therefore, this mechanism emulates fault effects (e.g., bus-access fault, cache fault, register-read fault, register-write fault) on the communication path between corresponding TLM models. Moreover, the communication protocol's safety requirements can be verified via fault injection before implementing the actual HW models.

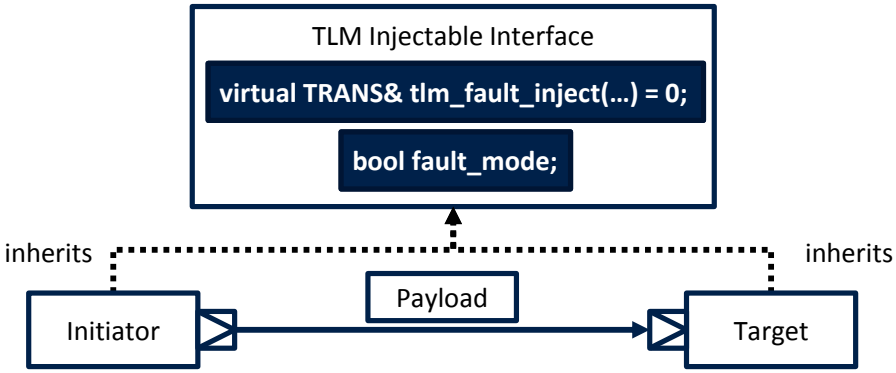


Figure 3.7.: TLM injectable interface

3.4.2. Injectable Payload

The injectable payload is an extended version of the TLM generic payload. It replaces the data types of each generic-payload attribute (e.g., address, command, data) with a FIO data type. Thus, each TLM payload attribute receives access

to dedicated fault-injection methods. The FIOs used for each attribute of the injectable payload are adapted to inject TLM faults (e.g., bus-access fault, cache fault, register-read fault, register-write fault). Even though they are more abstract than single-bit and multi-bit faults on RTL models and gate-level net-lists (e.g., stuck-at faults, bit flips), they are functionally equivalent and can be matched to each other.

3.4.3. Injectable Sockets

Based on the adopted TLM modeling style (i.e., untimed, loosely timed, approximately time), fault injection into TLM models may not offer sufficient accuracy. In the case of an approximately-timed simulation, for example, it may be difficult to schedule a fault injection into a TLM payload without prior knowledge of the correct TLM phase. Injectable sockets are developed to avoid such limitations. Injectable sockets use C++ function callbacks to connect and disconnect different fault models during a simulation (e.g., bus-transaction faults, delay faults from logic/arithmetic operations, instruction-fetch faults). The callbacks represent extensions of the TLM transport mechanism:

Pre/post-transport a fault injector connected to this callback injects faults before/after the execution of the transport method implemented in the *target* block. Therefore, faults injected with the *pre*-transport callback are useful for safety verification on the target block's forward path, whereas faults injected in the *post*-transport callback are used to emulate error-propagation from the *target* block on the TLM response path.

Transport override a fault-injector connected to this callback replaces the transport method implemented in the *target* block. Thus, faults injected here are particularly useful when the block's correct functionality must be temporarily exchanged with a faulty one, without replacing the code of the original *target* block.

TLM faults are injected by a fault injector which is connected to a specific callback either on the initiator or target socket (Fig. 3.8). Faults are inserted during a TLM transport call (e.g., b_transport, nb_transport_fw, nb_transport_bw). Injectable sockets allow fault models to directly access the transported payload's attributes, inject timing faults by modifying the TLM transport's delay argument, and probe the TLM-phase argument. Each fault injector contains a fault model

and a SystemC process. Fault models (e.g., bus-access fault, cache fault, register-read fault, register-write fault) are a part of the fault injector and not part of the analyzed TLM model. Fault models can be (dis-)connected during simulation to emulate the behavior of permanent or transient faults via a dedicated SystemC process. By developing fault models outside of the TLM models into which faults are injected, non-intrusive fault injection and fault modeling are possible. Furthermore, such fault models provide higher fault-model re-usability (e.g., register-address fault, bus-access fault).

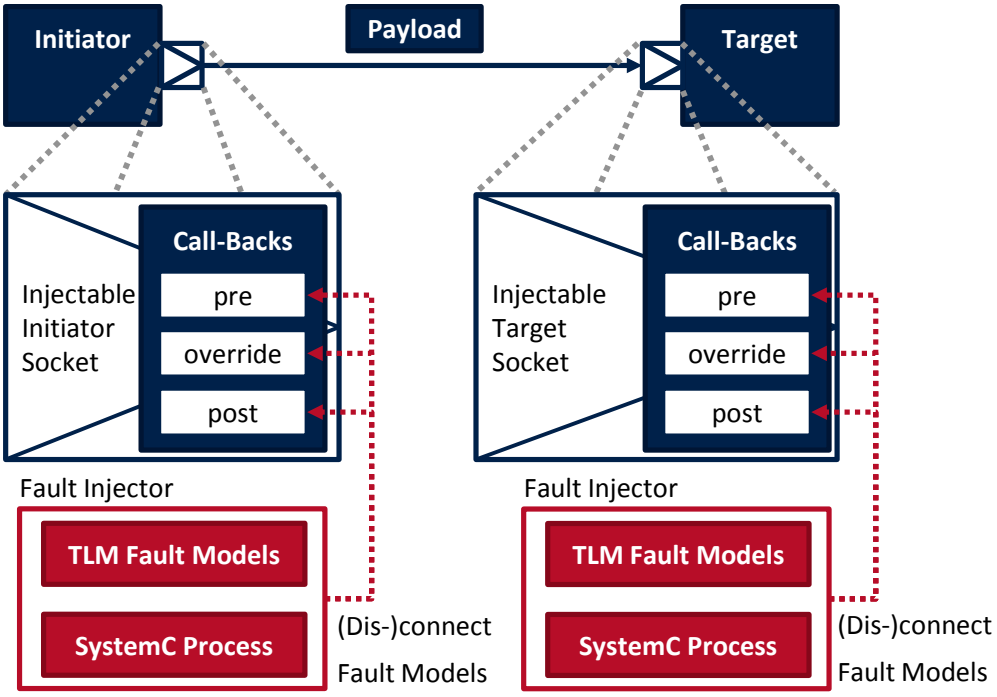


Figure 3.8.: TLM injectable sockets

Compared to FIOs which are internal components of a TLM model, sockets are found at the boundary between two or more TLM models. Consequently, injectable sockets only allow fault injection into the interface of these VPs, which is a useful method for fault injection when some models are only accessible as black boxes (e.g., third-party models). However, faults can only be injected at the interface between the black-box model and its test-bench or another TLM model. To also inject faults inside TLM models, injectable sockets may be combined either with SCFIT (see Chapter 3.2) or with FIOs (see Chapter 3.3). When using SCFIT, faults are injected at the boundary between two or more TLM models using injectable sockets and SCFIT is used to inject faults into the TLM models

themselves. Alternatively, the TLM model's structure may be enhanced using FIOs and faults may be injected from the test-bench by calling the appropriate FIO's fault-injection method.

3.5. Summary

The fault-injection tools mentioned above have led to a comprehensive fault-injection library for SystemC/TLM VPs. Each tool addresses a specific problem or a set of limitations present in the current scientific literature. For instance, they enable non-intrusive fault injection into SystemC/TLM VPs. In other words, fault injection into HW models occurs without changing the original model's source code or structure. SCFIT even allows non-intrusive fault injection into pre-developed (i.e., legacy) SystemC/TLM VPs. All tools support generic fault models (e.g., stuck-at-0, stuck-at-1, bit-flips) and the injection of single-bit and multiple-bit faults, which improves the fault-modeling flexibility of any safety-critical system. Furthermore, they enable modeling and reuse of abstract TLM fault models (e.g., bus-transaction faults, register-access faults) across multiple VPs as generic faults. As a result, safety experts can integrate these tools into safety-verification flows and use them for the safety verification of complex HW systems. The simulator-command extensions added to the SystemC/TLM libraries also enable VP integration into third-party models or test-benches without publishing the VP's source code. In this case, third party models or test-benches access the VP's fault-injection API. Hence, the intellectual property present within the integrated VP remains hidden.

4. Improving the Correlation of Fault-Injection Results

4.1. Introduction

Emerging simulation-based fault-injection methods have enabled research on the comparison of fault effects on different abstraction levels. However, results from fault injection into VPs correlate poorly with results from RTL and gate-level models [78], [79], [81]. In [79], [81], the poor correlation is blamed on the VP's high abstraction level. However, other reasons also exist, which lead to such a poor correlation. This chapter describes the fault-masking effects, which affect the propagation paths of injected faults, and introduces the notions of fault-matching points, pseudo-faults, and pseudo-failures. Furthermore, it introduces two generic and automated methods which guarantee correlation of VPs and gate-level net-lists.

4.2. Fault-Masking Effects

As already presented in Chapter 2.2.6, safety verification may be accelerated by continuously abstracting fault effects from one abstraction level to the next. However, in doing so, the safety-verification accuracy is gradually lost as each higher abstraction level contains less information than the previous one. This loss of accuracy is partly caused by the accidental loss of fault-masking effects through abstraction.

Fault masking is an intrinsic property of HW systems and contributes to the system's overall robustness against faults. Currently, four main fault-masking effects are known which influence fault propagation of random HW faults: electrical [44], latch window [44], temporal [147], and logical [44].

4.2.1. Electrical

Electrical masking effects are modeled at the circuit level, where a particle strike upsets a wire's voltage. The particle strike is modeled as an electric pulse. This pulse occurs randomly during system execution and is attenuated by the circuit's logic gates. The resulting soft error is considered masked if the pulse is sufficiently attenuated and does not cause a failure. However, if an input of a logic gate samples an electric pulse with a strong-enough amplitude, then the logic gate registers the faulty information as a bit-flip, and the soft error propagates. This effect cannot be modeled at the VP level since VPs do not model voltage pulses.

4.2.2. Latch Window

Latch-window masking is a particular case of electrical masking which occurs at the input of sequential elements and is characterized by the sequential element's setup and hold times. Particle strikes propagate from combinational circuits to a sequential gate only if they first latch on to the sequential gate. In other words, propagation occurs if the electrical pulse of a particle strike reaches the gate's forward latch at the same time as the clock transition or latching window. Otherwise, the soft error is considered masked at the latching window of a sequential element. Similar to electrical masking, this effect is neglected on VPs since it cannot be modeled.

4.2.3. Temporal

Temporal masking is an effect exclusively attributed to soft errors. Here, a soft error is masked if its effect is overwritten by the system before it has the chance to propagate to a sequential element. For instance, if a bit-flip occurs within a register, the fault only propagates if the faulty data within the register is read by a subsequent flip-flop or latch. However, if the faulty register is rewritten with correct data before the fault gets a chance to propagate, then the fault is considered temporally masked [147]. This effect can be modeled on the VP level.

4.2.4. Logical

Logical masking occurs at the input of combinational gates and refers to the gate's logical dominance. OR gates exhibit logic-'1' dominance while AND gates exhibit logic-'0' dominance. For instance, if one input of an OR-gate

has a logic-'1' value, a fault injected into the other input (either logic '1' or '0') is masked since the gate's output is not affected. This masking effect is usually modeled incompletely at the VP level since VPs contain much less implementation information than RTL or gate-level models. Logical masking does not depend on the integrated circuit's technology. However, it is considered the most challenging type of masking to model [44].

4.3. Pseudo-Faults and Pseudo-Failures

Pseudo-faults are single-bit or multiple-bit faults injected into VPs which lead to system failures but whose effects are not reproducible on the actual HW system. For safety-analysis reasons, ISO 26262 considers gate-level models to be a sufficiently-good approximation of the real HW system [5]. Therefore, pseudo-faults can also be defined as VP-based fault-effects which cannot be reproduced through injection of single-bit faults into gate-level models. **Pseudo-failures** are pseudo-fault effects observed at the outputs of VPs. The leading causes of pseudo-failures are structural differences between a VP and its gate-level counterpart (e.g., limited timing information, development of behavioral models opposed to structurally-accurate models). These differences result in fewer fault-masking effects on VPs.

Fault injection into VPs often leads to different results compared to fault injection into RTL models [79]. For instance, consider the Verilog-RTL model from Listing 4.1. The truth table of this circuit only has two possible binary values: '0000' and '1111'. Fault injection into the input bit of this model creates a change in the whole 4-bit output. Furthermore, single-bit fault injection into any of the output port's bits will lead to a situation from Table 4.1. Consequently, there is no single-bit fault injection capable of flipping two output bits simultaneously with this implementation. This result is only achievable through injection of 2-bit faults (Table 4.2). Nevertheless, a VP has no notion of such a structural constraint. Neither does it support the notion of single-bit or multi-bit fault injection. Fault models on VPs mainly describe an abstract faulty state. In this case, a 'special-logic' error can have any 4-bit value from Table 4.1 or Table 4.2. Thus, the simulation of any fault effect from Table 4.2 on a VP represents a pseudo-fault injection since its effects are not reproducible through single-bit fault injection on the particular Verilog-RTL model.

Upon closer inspection, however, it is still possible to modify the Verilog model's implementation in such a way as to remove some pseudo-faults. In

```
module special_logic ( i, o );

input i;
output [3:0] o;

assign o = { i, i, i, i };

endmodule
```

Listing 4.1: Verilog model which exhibits pseudo-fault effects

Table 4.1.: Single-bit failures observed on special_logic module

Input	Injected fault	Output Bit	Output Failure
0	Stuck-at-1	0	0001
		1	0010
		2	0100
		3	1000
1	Stuck-at-0	0	1110
		1	1101
		2	1011
		3	0111

Table 4.2.: Double-bit failures observed on special_logic module

Input	Injected fault	Output Bits	Output Failure
0	Stuck-at-1	1, 0	0011
		2, 0	0101
		2, 1	0110
		3, 0	1001
		3, 1	1010
		3, 2	1100
1	Stuck-at-0	1, 0	1100
		2, 0	1010
		2, 1	1001
		3, 0	0110
		3, 1	0101
		3, 2	0011

this case, the subset of the pseudo-faults mentioned above become real faults. Consider Schematic 1 (Fig. 4.1a) as the gate-level representation of the Verilog model. Schematic 2 (Fig. 4.1b) is behaviorally identical, but it contains two extra wire nodes (Listing 4.2). These wires are represented by the Verilog-RTL model called special_logic_schematic2 as n0 and n1. In this case, single-bit fault injection into n0 and n1 simultaneously cause 2-bit failures at [o0, o1] and [o2, o3], respectively. ISO 26262 classifies these 2-bit failures as common-cause failures. However, not all combinations of Table 4.2 are achievable using the topology from Schematic 2. Many potential pseudo-faults are still present (Table 4.3). Nevertheless, HW experts can develop other topologies, which address these remaining combinations. Furthermore, some topologies may even exist which are pseudo-failure free. However, such a modeling attempt merely increases the development complexity of HW systems. In other words, HW experts should not manually develop low-level abstraction models (e.g., RTL models and gate-level net-lists) just to ensure the validity of fault-injection experiments on VPs. Instead, structural information from a HW system's gate-level layout shall be automatically back-annotated to the VP when specifically required. This way, HW experts enhance VPs with missing gate-level information, fault-matching points, and fault-masking effects [108].

(a) Schematic 1 (b) Schematic 2

Figure 4.1.: Schematics of special_logic Verilog module

4.4. Fault-Matching Points

Even though VP-based safety verification benefits from fast simulation speeds, rapid modeling, quick debugging, and early availability, it has lower accuracy than safety verification on RTL and gate-level models. Since only temporal and logical masking effects can be modeled on VPs, safety-verification results obtained from VPs must be correlated with the results from circuit-level models

```
module special_logic_schematic2 ( i, o );

input i;
output [3:0] o;

wire n0; // Illustrated as red wires
wire n1; // in the diagram of Schematic 2

assign n0 = i;
assign n1 = i;

assign o = { n0, n0, n1, n1 };

endmodule
```

Listing 4.2: Modified Verilog model with fewer pseudo-fault effects than in Listing 4.1

Table 4.3.: Failures caused by 2-bit faults injected into special_logic_schematic2

Input	Injected fault	Output Bits	Output Failure
0	Stuck-at-1	2, 0	0101
		2, 1	0110
		3, 0	1001
		3, 1	1010
1	Stuck-at-0	2, 0	1010
		2, 1	1001
		3, 0	0110
		3, 1	0101

(e.g., gate-level net-lists). The goal of this correlation is the avoidance of pseudo-faults and the development of optimal safety mechanisms.

Fault-matching points are fault-injection locations which are available across two or more abstraction levels (e.g., VPs and gate-level net-lists). In other words, fault-matching points are elements of a HW model (e.g., signals, variables, ports) which are found on every abstraction level and which support fault injection. For instance, consider an RTL model and an equivalent gate-level net-list. These abstraction levels implicitly present a significant number of fault-matching points such as input-output interfaces, registers, and all signals present on the RTL model. However, most combinational gates and their interconnecting signals do

not represent fault-matching points. RTL models use abstract representations of combinational gates, whose exact structure only becomes clear after a logic-synthesis tool generates and optimizes the gate-level net-list.

However, VPs have much fewer fault-matching points with lower abstraction levels. For instance, gate-level net-lists describe processor components, such as arithmetic-logic units (ALUs), utilizing hundreds or even thousands of gates. However, VPs and even RTL models reduce ALU implementations to simple operations (e.g., addition, subtraction, shifting). Even in the case of complex adder architectures (e.g., carry lookahead, carry save), which have distinct implementations at the gate-level, VPs replace them by a simple plus-operator. Hence, the adder's structure becomes wholly abstracted, and any potential fault-matching points become limited to the adder's operands.

The following presents an example of the differences in fault-matching points across gate-level models and VPs on an abstract graph-based representation of a safety-critical SoC (Fig. 4.2a). The **nodes** represent components, such as flip-flops and combinational logic gates, which store system states. The unidirectional **edges** represent the flow of data across these system components. The VP is functionally equivalent to the gate-level net-list (Fig. 4.2b). Since VPs are more abstract than gate-level net-lists, they also contain fewer implementation details. These structural differences are represented by fewer nodes and a different configuration of edges.

(a) Gate-level net-list (b) VP

Figure 4.2.: Graph-based representation of missing fault-matching points across gate-level net-lists and VPs

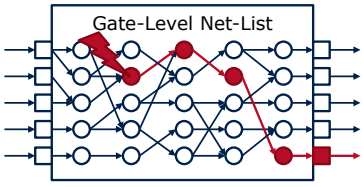
To understand why missing fault-matching points lead to the observation of pseudo-failures, the propagation paths of injected random HW faults must be analyzed. For this, the propagation path on the gate-level model is used as a reference (i.e., the red path in Fig. 4.3a). In this example, the VP is missing the fault-injection location used for gate-level fault injection. In this case, faults must be injected into different system locations, which will then follow the same propagation paths as on the gate-level model. However, manual inspection of

the system's structure and its communication path is necessary to determine correct alternative fault-injection locations. In the example above, the following fault-injection scenarios become apparent on the VP:

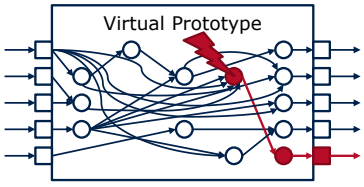
Scenario 1 inject a fault closer to the VP's output (Fig. 4.3b).

Scenario 2a inject a fault closer to the VP's input (location a) (Fig. 4.3c).

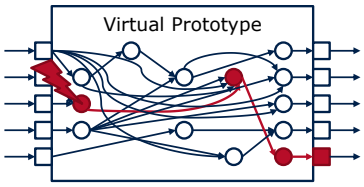
Scenario 2b inject a fault closer to the VP's input (location b) (Fig. 4.3d).



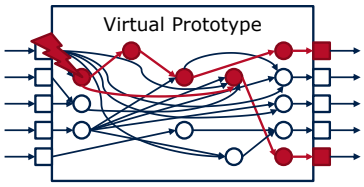
(a) Reference fault-injection outcome



(b) Scenario 1: **after** the missing fault-injection point; **same outcome**



(c) Scenario 2a: **before** the missing fault-injection point; **same outcome**



(d) Scenario 2b: **before** the missing fault-injection point; different outcome (**pseudo-failure observed**)

Figure 4.3.: Graph-based representation of fault-propagation paths on gate-level net-lists and VPs

In all three scenarios, the same system failure as on the gate-level net-list is observed. However, in the third scenario (Fig. 4.3d), a pseudo-failure is also observed because the fault-injection location contains an extra fault-propagation path. Therefore, the selection of the alternative fault-injection location is essential. Unfortunately, manual analysis of complex systems is not feasible, especially when multiple scenarios must be considered when selecting alternate fault-injection locations. For this reason, an automated method is needed which guarantees sufficient fault-matching points across VPs and gate-level net-lists, and avoids analysis of pseudo-fault effects on VPs.

4.5. Augmentation of Virtual Prototypes with Gate-Level Data

This section introduces two methods (i.e., VERITAS, VERITAS++) which avoid the injection of pseudo-faults into VPs, automatically link VPs to gate-level net-lists, and ensure sufficient fault-matching points to gate-level net-lists.

4.5.1. VERITAS

VERITAS (Verilog net-list to SystemC transformer) has been explicitly created to enhance VPs with missing fault-matching points from the gate-level net-list. While other approaches attempt to link VPs to RTL or gate-level models manually [81] or through co-simulation [82], VERITAS follows an automated approach which adds gate-level granularity to SystemC VPs.

VP Augmentation Process

VERITAS focuses on the abstraction of combinational gates to the VP level. It achieves this by using Python-based Verilog and Liberty-file parsers to extract the structure of combinational circuits and the functionality of the instantiated cells (i.e., logic gates) (Fig. 4.4). These extracted data are stored into a data model. Afterward, VERITAS arranges the extracted logic gates into topological order. Next, single-bit wires are directly transformed into Boolean variables (Fig. 4.5). Input buses are modeled as (long) integer types and are broken down into individual variables using **shifting** operations. Output buses are concatenated from different variables using **masking** operations (Fig. 4.6). Finally, combinational gates are transformed into logic operations, which respect the circuit's topological order of execution (i.e., mimic the original net-list's behavior). The result of this transformation is a C++ model with the same granularity as the original Verilog gate-level net-list but with higher execution performance (see Chapter 6.8.1).

VERITAS also generates SystemC and TLM wrappers for the resulting C++ code. The SystemC generator creates input and output ports and a SystemC process for the generated function. The SystemC process is sensitive to the model's inputs. The TLM generator creates a TLM target module with the blocking-transport method. The generated C++ code is then called directly by the transport method. The transport method's delay attribute is user-defined and may be set after code generation. The TLM generator only requires support

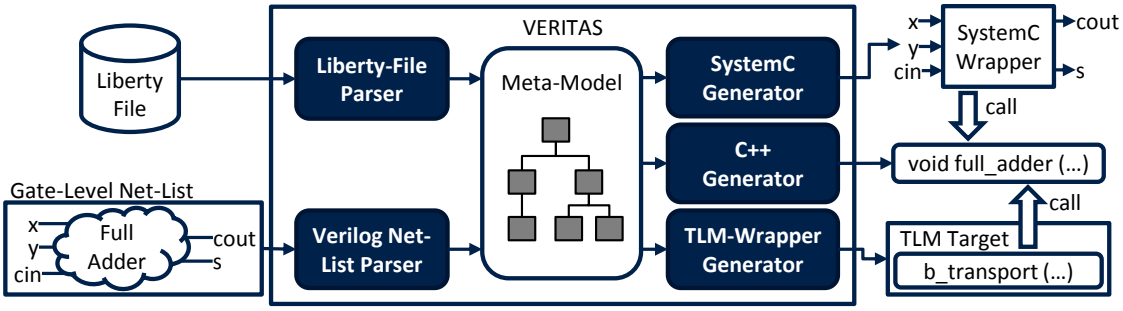


Figure 4.4.: VERITAS flow diagram

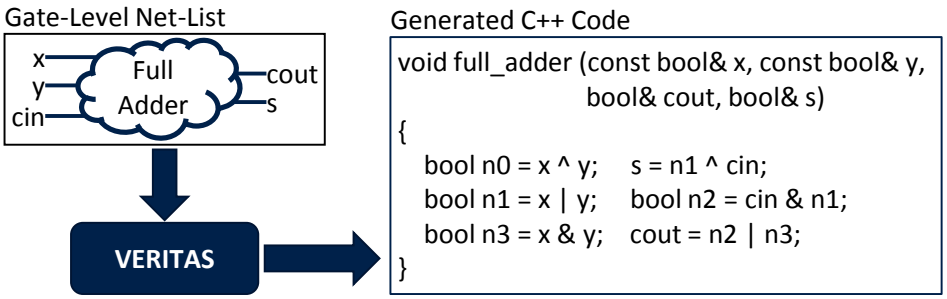


Figure 4.5.: VERITAS C++ representation of a full adder

for loosely-timed TLM models. As previously mentioned, combinational logic gates are generated as logic operations and are executed when one input of the logic cone changes its value. Afterward, the result is propagated to connecting SystemC/TLM models or (sub-)system output ports. Thus, entire logic cones (e.g., an adder module or a processor's ALU) are abstracted to a single-step method call which returns the logic cone's output (e.g., the adder's sum and its carry bit). Consequently, it is not required to generate approximately-timed TLM models (e.g., non-blocking-transport methods and TLM phases).

Gate-level information may be directly integrated into existing SystemC/TLM-based VPs by replacing behavioral functions modeled in TLM with code generated by VERITAS. Consider a HW system which requires a specific adder implementation (e.g., carry-save adder), for instance. On the VP, the adder may have been merely modeled using the '+' operator from C++. In this case, VERITAS can be used to augment the TLM model with the adder's gate-level information. The TLM '+' operation is then replaced with C++ code generated by VERITAS, which implements the HW model of an adder circuit. As a result, valuable fault-matching points are added to the VP since the generated code has the same granularity as the gate-level net-list. Furthermore, the TLM model is

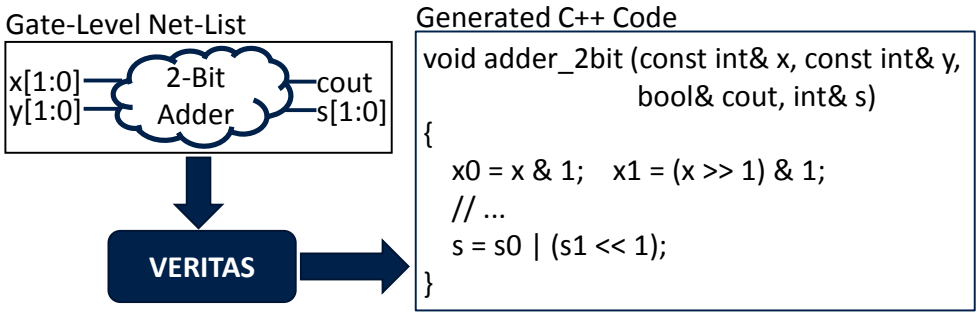


Figure 4.6.: VERITAS breakdown of multi-bit operations

enhanced with gate-level fault-masking effects, which ultimately improves the VP's fault-propagation paths during fault-injection simulations. Consequently, faults injected into the augmented VP have a 1-to-1 mapping to the gate-level net-list.

From a modeling perspective, the transformation of Verilog code into C++ code may not seem like an abstraction at all. However, this is not the case. Consider a HW system modeled using HDLs. First, HDLs use specific constructs to model the system's structure (e.g., modules, processes, signals, ports). Second, HDLs use particular modeling methods (e.g., process-sensitivity lists) to emulate the parallel behavior of HW elements. Third, systems are modeled differently from one abstraction level to the next: (i) gate-level uses wires and gates, (ii) RTL uses buses and registers, and (iii) VPs use sockets, payloads, and variables. Each abstraction level focuses on modeling system behavior while removing specific properties which more accurately describe the system's structure. VERITAS performs the same process. VERITAS abstracts Liberty files which describe a gate's structural, behavioral, and electrical properties into Verilog models. As a result, the gate's electrical properties are ignored (i.e., abstracted out). VERITAS further abstracts the system by transforming combinational HDL blocks into C++ code (with optional SystemC and TLM wrappers). In this case, it is not the system's structure which is abstracted but its representation and thus, its simulation semantics. Combinational gates are no longer described by modules and processes but by C++ variables and simple logic operations. Furthermore, gates are ordered topologically and simulated sequentially instead of using event-driven simulation. Thus, process-sensitivity lists which are mainly present in HDLs are also abstracted out. After augmenting the VP with gate-level information, new fault-injection locations become available on the TLM model which require verification. Consequently, the VP's corresponding test-bench must be extended with fault-injection test cases.

Fault Injection

Fault injection may be performed with any tool presented in Chapter 3. SCFIT can be used out of the box on the C++ models generated by VERITAS [108]. Furthermore, SCFIT's configuration is significantly simplified for each fault-injection scenario because the C++ variables created by VERITAS are not templatized and do not have read/write methods (Fig. 4.5). Thus, SCFIT only requires one HW breakpoint on the generated C++ method and one HW watchpoint on each C++ variable targeted for fault injection. Additionally, simulator commands for SystemC (i.e., FIOs) and TLM (i.e., injectable sockets) are added to VERITAS by extending its C++ and TLM-wrapper generators. In the case of FIOs, the original Boolean variables are replaced with global FIO instances, whose fault-injection methods can be accessed directly from the system's test-bench (Fig. 4.7) [112].

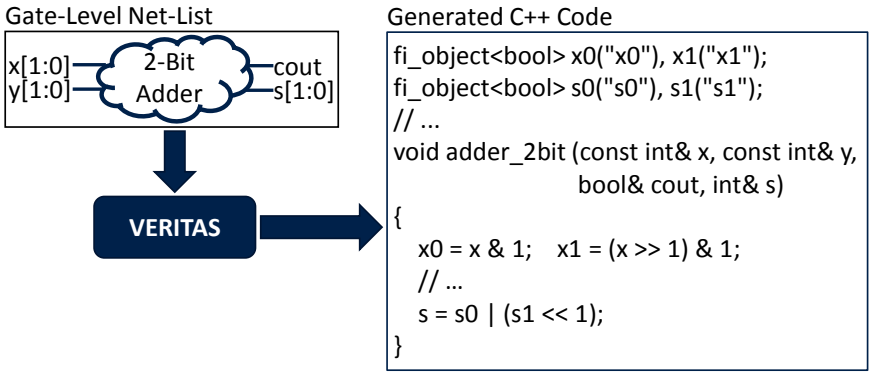


Figure 4.7.: Extension of VERITAS' C++ generator to support fault-injection objects

4.5.2. VERITAS++

VERITAS++ is an extension to VERITAS. The Python-based model-driven approach from VERITAS is replaced in VERITAS++ by a C++-based tool called Verilator [148] (Fig. 4.8). Compared to VERITAS which only processes combinational-logic blocks within a HW system, VERITAS++ also processes sequential-logic blocks (e.g., flip-flops, latches). Consequently, VERITAS++ represents a safety-verification framework suitable for safety verification on complete HW systems.

Verilator requires synthesizable Verilog models for each gate described in a Liberty file. As a result, VERITAS++ is used to parse Liberty files and transform them into functional Verilog models (Fig. 4.8). The gate models generated by VERITAS++ are then integrated into the system's gate-level net-list, parsed

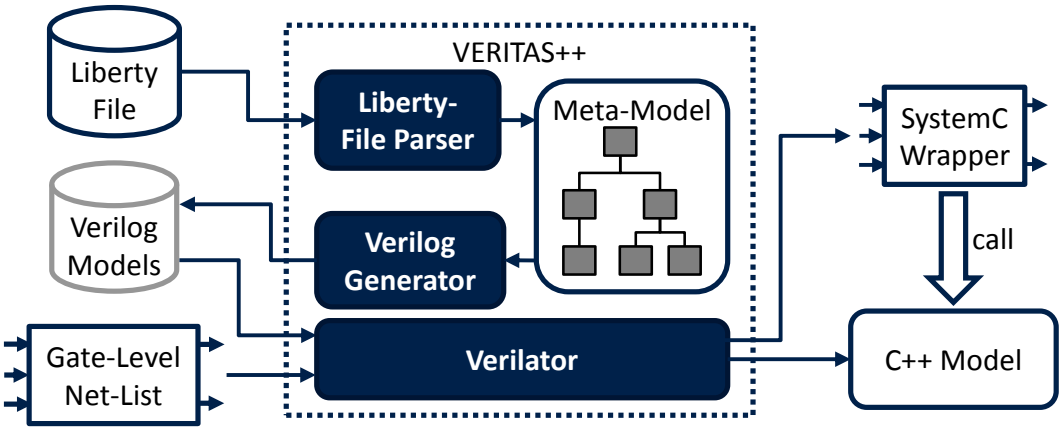


Figure 4.8.: VERITAS++ flow diagram

by Verilator, and abstracted into SystemC/C++ code. Consequently, VPs no longer need to be augmented with gate-level information. Instead, gate-level net-lists are directly abstracted to C++ and SystemC models. Similar to VERITAS, Verilator abstracts the target system by transforming the known HDL structure into C++ code (with an optional SystemC wrapper).

Abstracting Gate-Level Net-Lists to Virtual Prototypes

Verilator is an open-source Verilog to C++/SystemC compiler, which transforms synthesizable Verilog and a subset of SystemVerilog code into functionally equivalent C++ or SystemC code (Fig. 4.9). Similar to VERITAS, Verilator levelizes the hierarchy of a Verilog model and optimizes (i.e., collapses) redundant signals within the HW model. All HW signals, including those removed during optimization, are traced using VCD files.

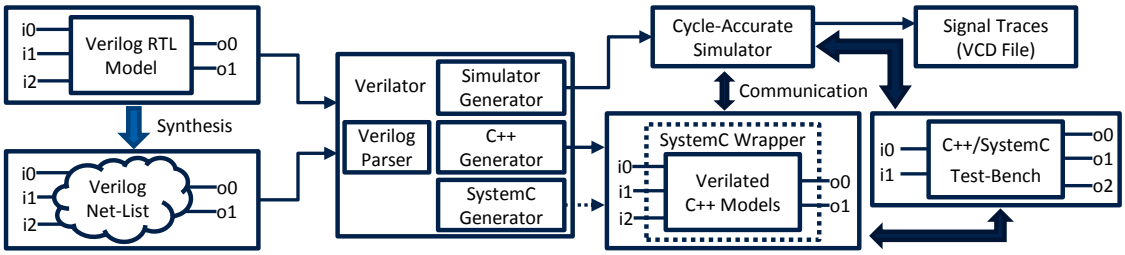


Figure 4.9.: Verilator flow diagram

Generated SystemC code represents a SystemC module with corresponding SystemC-specific input and output ports. Furthermore, the transformed code is

executed by a SystemC process which is sensitive to the module's inputs. The SystemC module is a wrapper around the levelized C++ code (Fig. 4.9). The generated code can be directly compiled with a C++ compiler (e.g., Clang, G++).

Verilator provides a cycle-accurate compiled-code simulator for the generated levelized code. Compared to EDSs, which schedule and execute processes based on event triggering (e.g., write events on signals), Verilator executes the whole design in topological order at every simulation step. As a result, Verilator's simulator utilizes only a small number of internal state variables (i.e., event queues are not needed since no event scheduling takes place during simulation).

Fault Injection

Verilator does not support fault injection. The authors of [78] have used Verilator for safety verification by transforming RTL code into SystemC/C++ models and by connecting these models to TLM test-benches. Faults have been injected into the TLM models, and their impact has been analyzed on the transformed RTL models. However, this approach requires TLM models with fault-injection capabilities and also needs synchronization between the TLM test-bench and the generated SystemC/C++ models.

VERITAS++ uses a different fault-injection approach to that proposed in [78]. VERITAS++ extends Verilators's open-source C++ and SystemC generators to add support for simulator commands (i.e., FIOs). Since Verilator uses C++ data types to store data from HW models which do not have dedicated fault-injection capabilities, simulator commands have been added by replacing these C++ data types with FIOs. As a result, all HW signals become instances of FIOs. Hence, faults can be directly injected from a C++ or SystemC test-bench by merely calling the appropriate fault-injection method. Since FIOs can be optimized by the C++ compiler, they offer better performance than SCFIT's GDB-based approach. Furthermore, as already mentioned in Chapter 3.2, GDB's performance is limited by the number of HW breakpoints and watchpoints available in current processor technologies.

4.6. Summary

Safety experts perform fault injection into VPs to establish a system's fault tolerance early in the HW system's development cycle when architectural errors are easy to correct by implementing better safety mechanisms. However, the lack of implementation details, fault-matching points, and fault-masking effects on

VPs leads to the observation of pseudo-failures. In turn, safety verification on VPs becomes difficult if not even impossible. Consequently, safety verification must be repeated on the RTL or gate level to obtain correct results. To address this issue, two methods (i.e., VERITAS and VERITAS++) have been introduced in this chapter, which improve the correlation of fault-injection results across gate-level net-lists and VPs. VERITAS augments existing VPs with combinational information from gate-level net-lists by transforming this information into C++ code. VERITAS++ further transforms entire gate-level net-lists (i.e., sequential and combinational information) into VPs.

5. Optimizing Fault-Injection Simulations

5.1. Introduction

Fault-injection campaigns require a considerable number of simulations to provide accurate results. Currently, SFI methods using Monte-Carlo simulations must exercise tens of thousands of fault-injection experiments on a safety-critical SoC to achieve results with sufficient statistical confidence such as 99.8%. Furthermore, these simulations must be repeated on multiple SoC workloads (i.e., real-life applications). Thus, complex SoCs may require millions of fault-injection simulations.

Moreover, some injected faults may be masked by the simulated system due to its inherent fault-masking effects. In this case, the simulation does not provide any relevant information for two reasons. First, the system may be safe against the effects of the injected fault, in which case, the fault may be excluded from the fault-injection campaign altogether. Second, the masked fault may lead to a system failure under different simulation circumstances such as using another system workload or changing the fault-injection attributes (i.e., time, fault type, and location). Hence, from a verification perspective, the simulation time has been wasted. It would be more efficient to determine beforehand if the analyzed system is immune to the effects of a specific fault or if the injected fault would lead to fault propagation during a simulation. Such information would drastically reduce the number of simulations required to determine a system's fault-tolerance levels. However, pure SFI methods do not use such information about the system.

To reduce the number of fault-injection simulations, this chapter introduces a safety-verification flow called SaVer, which enables automatic fault injection into VPs. Furthermore, this chapter introduces four optimization techniques, developed to reduce the verification complexity of safety-critical SoCs and speed up fault-injection simulations: (i) removal of redundant fault-injection locations,

(ii) discovery and removal of fault-injection simulations which do not lead to fault propagation (iii) parallelization of fault-injection simulations, and (iv) simulation checkpointing.

5.2. Measures for Verification Completeness

A system's fault-tolerance level may be accurately determined using formal (mathematical) or statistical (simulation-based) verification methods. Formal methods employ mathematical properties, also known as assertions, to exhaustively analyze HW models [149], [150]. As a result, they automatically check all possible fault-injection scenarios and fault-propagation paths. Therefore, they have an essential measure for completeness. Unfortunately, formal methods are computationally expensive and require a long time to complete their analysis even for relatively small systems. Consequently, formal methods are not yet scalable with sizable safety-critical SoCs.

Statistical methods analyze fault effects by randomly injecting faults into a system (i.e., one fault per simulation) using Monte-Carlo simulations with uniform distributions [102]. In other words, the same probability is used to randomly inject faults into any system location and at any simulation time. However, contrary to formal methods, SFI methods do not have an implicit measure for completeness. Therefore, it is important to determine how many fault-injection simulations are required to reach a sufficient confidence level. In [102], the number of fault-injection simulations is expressed based on the statistical distribution of faults within a system:

$$n = \frac{N}{1 + e^2 \frac{N-1}{t^2 \cdot p \cdot (1-p)}} \tag{5.1}$$

where

- n = the number of fault-injection simulations
- N = the number of faults extracted from the system's fault-verification space
- p = the proportion of faults from the fault-verification space which can lead to failures. Conservative SFI methods assume $p = 0.5$.
- e = the margin of error
- t = the standard error obtained from N (i.e., the t-test). This value depends on N and the expected confidence interval of SFI results.

This equation can be more easily visualized by plotting it using the following example values: $p = 0.5$, $e = 1\%$, and $t = 3.0902$ (Fig. 5.1). This t value

corresponds to a confidence interval of 99.8% as defined in [151]. The increase of N leads to the increase of n. In other words, the more faults a system's fault-verification space presents, the more fault-injection simulations are required to analyze the system sufficiently. Nevertheless, the graph also presents a horizontal asymptote when N tends to infinity. Hence, only a limited number of fault-injection simulations is required to analyze a system regardless of the size of N sufficiently.

The maximum number of fault-injection simulations n_{max} needed by a HW system with large fault-verification space can be calculated from Eq. 5.1 as:

$$n_{max} = \lim_{N \to \infty} f(N) = \frac{t^2}{e^2} \cdot p \cdot (1 - p) \tag{5.2}$$

By inserting the previous example values for p, e, and t into Eq. 5.2 and by rounding up to the next integer, n_{max} becomes 23 874. In other words, a conservative SFI approach (i.e., $p = 0.5$) requires only 23 874 fault-injection simulations with different faults to sufficiently analyze a HW system. The SFI results have a 99.8% confidence interval and 1% margin of error regardless of the size of the system's fault-verification space.

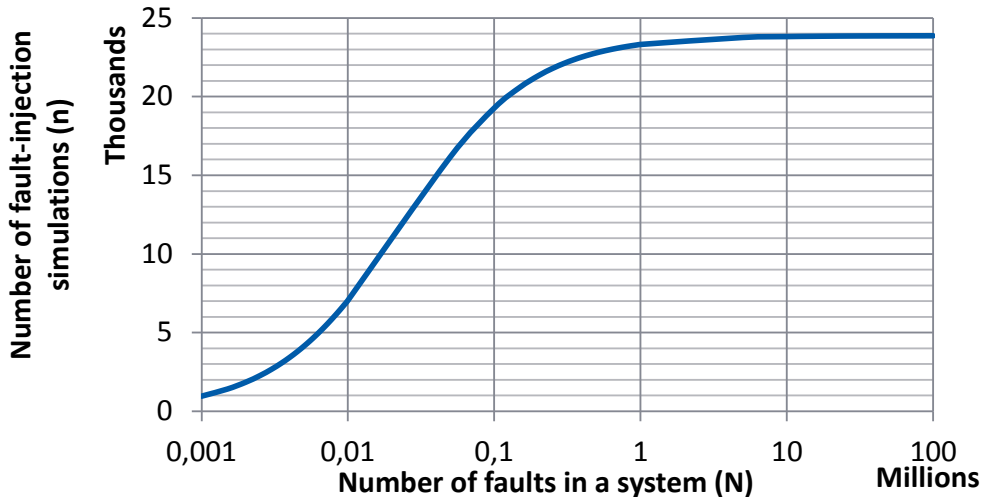


Figure 5.1.: Saturation curve for the number of fault-injection simulations n given by the number of faults N ($p = 0.5$, $e = 1\%$, and $t = 3.0902$ for 99.8% confidence interval)

5.3. SaVer

This thesis introduces SaVer, a simulation-based safety-verification flow for SystemC/TLM VPs, which allows statistical injection of tens of thousands of faults during a fault-injection campaign.

5.3.1. Fault-Injection Flow

SaVer uses a three-phase approach to inject faults into a VP and determine the results of fault-injection simulations (Fig. 5.2). First, a reference simulation is run with a given workload and without fault injection. Simulation results are saved as signal traces provided by the simulator. Next, multiple fault-injection simulations are executed using the same initial workload. Each fault-injection simulation injects a different fault into the HW model. Finally, fault-injection simulation results are compared to the results obtained from the reference simulation.

Mismatches in the simulation outputs are the result of fault propagation through the system. If mismatches are observed in any of the system's outputs, a failure is recorded. If no mismatch is found, the fault is considered masked by the system. In this case, improved testing is required to determine if the fault is safe or if the injected fault's effect is latent. Under the correct conditions (i.e., system workload, simulation duration, and fault-injection attributes) latent faults may lead to failures.

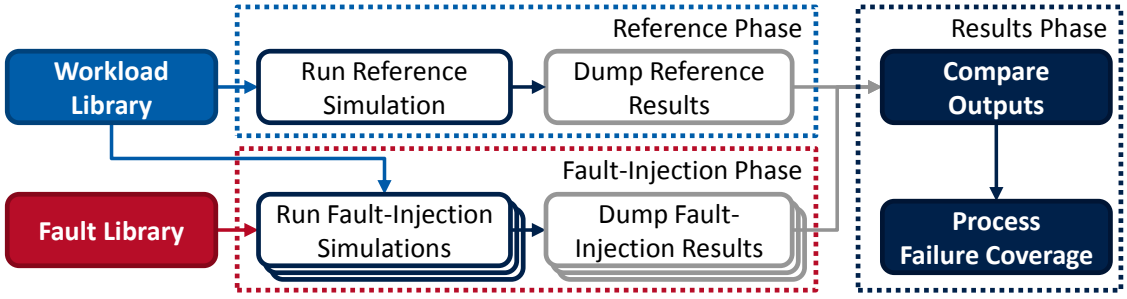


Figure 5.2.: SaVer's fault-injection regression flow

5.3.2. Fault-Injection Methods

SaVer utilizes the three methods introduced in Chapter 3 to generically inject faults into the analyzed VPs. SaVer supports two fault-injection regression modes:

SFI and user-defined fault injection (UDFI). During **SFI**, fault-injection locations are selected randomly from the system's list of internal signals. In this case, only one fault is injected per simulation. Fault types (e.g., stuck-at-0, stuck-at-1, bit flip) are also randomly chosen. Permanent faults are injected at the beginning of a simulation, and their effects last for the whole simulation. Transient faults are injected at a random simulation time and last until overridden by the system. Consequently, a system's average fault tolerance can be computed. During **UDFI**, all fault attributes (i.e., location, type, simulation time) are user-defined. This mode is mainly designed for debugging and directed testing. In this case, users can inject any number of faults into the analyzed system. Fault attributes which result from SFI or UDFI are stored in a fault library. SaVer uses fault libraries to inject faults during fault-injection simulations.

5.4. Spatial and Temporal Fault Pruning

This section introduces two methods to remove redundant fault-injection locations from a system. The first method implements spatial fault pruning. The latter utilizes temporal fault pruning.

5.4.1. Removal of Redundant Fault-Injection Locations

When abstracting gate-level net-lists to SystemC/C++ models using VERITAS++, Verilator's optimization techniques significantly reduce the number of signals present on the net-list. These optimization techniques remove redundant signals which typically have the same behavior during simulation. An excellent example of redundant signals is output ports of a subsystem connected to input ports of other subsystems through signals. The triplet (output port, signal, input port) can be collapsed to a single C++ variable. Nevertheless, for debugging purposes, Verilator still traces all signal values (including the ones removed).

Since the optimized signals are redundant, fault injection into any of them results in the same fault propagation. In other words, fault effects observed on these signals are equivalent. Thus, it is sufficient to inject faults into any of the redundant signals to obtain accurate results. Spatial fault pruning does not reduce the number of fault-matching points across generated SystemC/C++ models and their equivalent gate-level net-lists. It only removes redundant fault locations and optimizes the fault-verification space.

5.4.2. Simulation-Trace Analysis

Fault-effect analysis can be accelerated by pruning the number of faults available for injection. Fewer faults to inject also means fewer simulations to run. However, it is essential to prune only those faults which, after injection, do not propagate through the system. This thesis introduces a fault-pruning method for SaVer, which determines safe faults and groups redundant faults with the same propagation paths. This method has been developed for VPs based on SystemC and TLM .

Simulation-trace analysis inspects read and write accesses on components of a HW system to determine whether faults injected into those components can propagate through the system. Before running a fault-injection simulation, the analysis determines if the system can mask an injected fault. In this case, this fault can be excluded from a fault-injection campaign because its effect becomes overridden by the system. However, if the fault is not masked by the system, a complete fault-injection simulation is needed to determine whether the fault's effect reaches the system's output and becomes a failure.

SaVer's fault-pruning method follows three steps: (i) analysis of simulation traces, (ii) extraction of valid fault-injection intervals, and (iii) fault-library generation.

Simulation traces are usually dumped into a trace file, such as a VCD. However, typical HDL simulators only offer information about write accesses on HW components. Since lifetime analysis also requires knowledge about read accesses, this thesis introduces a new trace-file format, called value-access dump (VAD). The VAD format efficiently extends the VCD format with read-access information (Fig. 5.3). Similar to VCD, VAD files record value changes on each traced signal at every simulation time stamp. Additionally, VAD files record the type of access (i.e., read or write) and the number of accesses per simulation time.

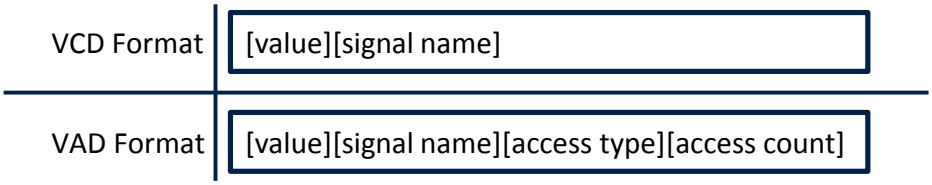


Figure 5.3.: VAD format compared to VCD format

To analyze simulation traces, a fault-free simulation is first performed, and a VAD file is dumped. Next, the resulting VAD file is parsed using a Python-based VAD-file parser. Then, an algorithm is applied to the extracted data to determine

which faults are masked. Finally, after eliminating all maskable faults, a fault library is generated and used in subsequent fault-injection simulations.

SaVer's fault-pruning algorithm scans all read and write accesses issued by the HW simulator on each HW register (Fig. 5.4). Fault masking occurs when faults are injected before a write access since transient faults (e.g., bit-flips) are overridden by the HW system. However, fault injection before a read access always leads to fault propagation (at least for a short distance) since the register's faulty value is read by a different register or logic gate. The propagated fault may be masked later during system execution or, under appropriate circumstances, may lead to a system failure.

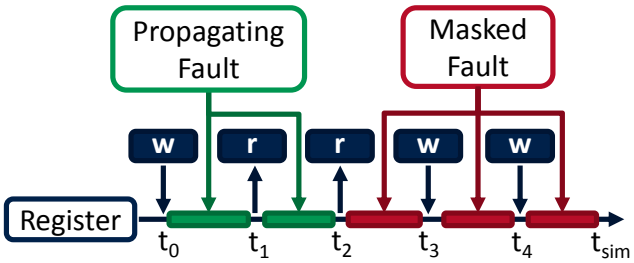


Figure 5.4.: Difference between masked faults and faults which propagate based on read-write accesses on registers

After determining which faults propagate through the system, this algorithm further removes faults which have the same propagation paths. For instance, consider an arbitrary workload which writes a generic register 100 ns after the simulation starts. Additionally, the system reads the register's content at 150 ns. Any fault injected anywhere within the 100 ns and 150-ns time stamps (e.g., at 101 ns, 115 ns) will ultimately have the same fault-propagation path. As a result, these faults are equivalent, and can be injected as a single fault. Thus, this method further reduces the system's fault-verification space.

5.5. Parallelization of Fault-Injection Simulations

One method by which SaVer reduces the duration of fault-injection campaigns is by executing simulations in parallel. This is achieved because SaVer's fault-injection simulations run independently from each other. Thus, instead of simulating each fault injection sequentially, they can be run in parallel by using a server farm (computer cluster). When using VERITAS++, fault-injection simulations are run after compiling the C++ and SystemC models generated

by Verilator. Fault-injection simulations are started in parallel using a Makefile which controls the compilation and simulation of the C++ code [152].

5.6. Simulation Checkpointing

The simulation time of each fault injection can be reduced by using checkpointing mechanisms. By saving simulation snapshots, the simulation time leading up to the insertion of a HW fault can be avoided. As a result, the duration of a fault-injection simulation may be limited to analyzing the propagation path of a fault through the safety-critical HW system.

A checkpoint-restore approach has been developed for C++ and SystemC models generated by Verilator and has been integrated into SaVer's fault-injection flow. In other words, this approach is designed for VPs with gate-level granularity. The approach follows the standard two steps of snapshotting: checkpointing a simulation and restoring a simulation from a previously saved checkpoint.

5.6.1. Checkpoint

Checkpoints are generated in two steps: simulation (including signal tracing and trace generation) and then, checkpoint generation. In the first step, a simulation is run until a predetermined simulation time. During this time, the HW model's activity (e.g., signal and register values), internal variables of Verilator's CAS, and simulation time are stored for each simulation time step. In the second step, the previously gathered information is used to generate an arbitrary number of checkpoints offline.

Checkpoints are not generated as a single file like in other snapshotting approaches (e.g., [93]–[95]), but broken down into one header file and multiple data files (i.e., one per checkpoint). The header file contains a list of signals and registers saved from the HW model. The data file includes the saved signals' values listed in the same order as their names. Signal names contain the full hierarchy of the HW model and thus require several orders of magnitude more storage space than their values whose size depends only on their bit-widths. Moreover, signal names are constant over all checkpoints, whereas their values may vary from one checkpoint to the next. Thus, saving signal names within every checkpoint file represents a sub-optimal usage of limited hard-disk space. By separating a HW model's signal names and values into two files and by only generating new files for the signals' value changes, the size of checkpoints

is optimized when generating large numbers of checkpoints. Hard-disk space requirements of checkpoints are further reduced by only snapshotting Verilator's optimized signals.

While the HW model's activity is saved in the form of a VCD file generated by Verilator, the internal CAS variables are not tracked by Verilator's VCD mechanism. For this reason, Verilator's C++ generators have been extended to monitor internal states of Verilator's CAS as well. Thus, internal simulation states are saved using a comma-separated value (CSV) format, consisting of triples of internal variables' name, value, and the current simulation time stamp.

Snapshots are created using a model-based Python application which generates checkpoints for a list of simulation-time values (Fig. 5.5). First, a list of optimized signals generated by Verilator, the HW model's VCD file, and the internal simulator variables saved in CSV format are parsed. The VCD-file parser is described in [153]. Next, the parsed data are stored into a data model from which the values of each optimized signal and each internal simulator variable are extracted at a given simulation time. Finally, the extracted data are stored as name-value pairs, which are compressed and written into the checkpoint's header file (i.e., signal and variable names) and data files (i.e., signal and variable values). When generating multiple checkpoints, the header file is generated only once, while a data file is generated for each checkpoint.

Figure 5.5.: VCD-based checkpointing-generation flow

The VCD-file parser uses the model-based Python application from [153]. This application's data model is structured based on the VCD format's ASCII symbols (Fig. 5.6); each symbol contains a list of time-value pairs and a list of associated nets (i.e., signals). Each time-value pair contains the signals' value paired with

its corresponding simulation time. Each associated net (i.e., HW model signal) contains four attributes: name, bit-with, type, and hierarchical name within the HW model.

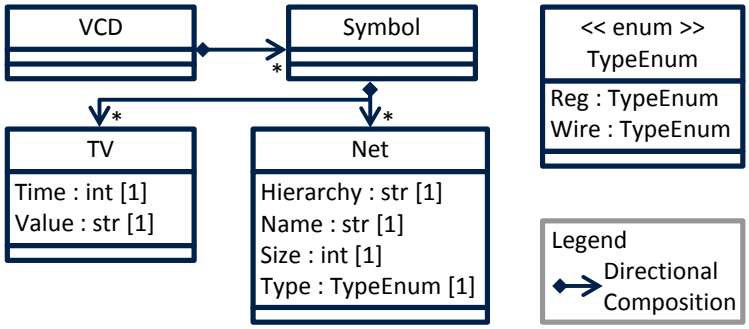


Figure 5.6.: UML diagram of meta-model structure

The appropriate values of nets and internal simulator variables are extracted by iterating through the VCD's data model and searching for the net's value at a given simulation-time. If a net does not explicitly contain an entry for the probed simulation-time value, the first simulation-time entry smaller than the given one is selected from the list (i.e., the net's value was set at a previous simulation time and was not changed since). By only saving the HW model's activity and the internal states of Verilator's CAS, each checkpoint's hard-disk space requirements are significantly reduced. Moreover, the creation of checkpoints after a simulation allows better checkpoint-lifetime management. In other words, checkpoints can be generated in chunks when needed and then deleted to save-up hard-disk space; this allows the generation of other checkpoint chunks from the same reference simulation. Thus, hard-disk space management is more easily controllable, and fewer simulations are required for generating checkpoints.

5.6.2. Restore

The initialization phase of Verilator's simulator is modified to allow restoring a simulation from a saved checkpoint. Previously, this phase was used to initialize signals before the start of a simulation. However, when restoring from a checkpoint, the simulator must open the generated checkpoint header and data files and automatically initialize the simulator's internal variables as well as the HW model's signals (Fig. 5.7).

This restoring mechanism presents high flexibility, which allows it to be reused even if the HW model generated by Verilator was changed (e.g., signals added or

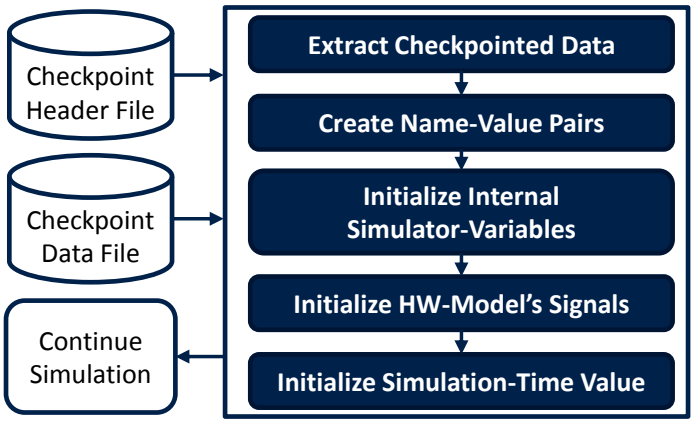


Figure 5.7.: VCD-based restoring-from-checkpoint flow

removed). This approach is highly beneficial to speed-up debugging (e.g., bug fixing of HW blocks independently of the checkpointed data), a feature which is currently unavailable in commercial HDL simulators. If signals are removed from the design, the restoring phase just ignores them. If a signal is added to the HW model, it can be manually appended to the checkpoint files as a new name-value pair. When restoring the simulation, the added signal's value before restoration is set to *undefined*; then, its value is restored from the checkpoint file and henceforth, updated by the simulator until the simulation ends.

5.6.3. Checkpointing within SaVer

To reduce the simulation time of fault-injection simulations, SaVer has been enhanced with checkpointing capabilities by tracing system states and internal simulator states for each reference simulation (Algorithm 1). The traced information is saved in the form of VCD files. Furthermore, a fault library is used to create a set fault-injection simulation times. Next, this set of simulation-time values and the reference VCD files are used to generate checkpoints. Finally, fault-injection simulations are run for all faults in the fault library by automatically restoring the reference simulation from an optimally chosen checkpoint (e.g., one clock cycle before fault injection). Compared to SaVer's original implementation (Fig. 5.2), the introduction of the VCD-based checkpoint mechanism adds an intermediary checkpointing phase (Fig. 5.8).

Algorithm 1 Checkpoint-driven fault-injection simulation

Require: workload library **and** fault library
Ensure: fault-injection simulation results
  for all workloads **do**
    Run reference simulation
    Dump VCD file with traced system states and internal simulator states
  for all fault-injection times **in** fault library **do**
    Create set of checkpoint-time values
  for all checkpoint-time values **do**
    for all pairs of reference VCD files **do**
      Create checkpoint
  for all fault-injection times **in** fault library **do**
    Run fault-injection simulation from optimal checkpoint
    Dump fault-injection VCD file

5.6.4. Performance Analysis

The maximum simulation speed-up, which can be reached using simulation checkpointing, can be mathematically described as:

$$S = \frac{N_{faults} \cdot t_{sim}}{\sum_{i=1}^{N_{faults}} (t_{sim} - t_{chkpt_i})} \tag{5.3}$$

where

S = speed-up factor
N_{faults} = number of faults to be injected during a fault-injection campaign
t_{sim} = number of simulation steps executed per fault-injection simulation
t_{chkpt_i} = simulation time saved after restoring a simulation from a checkpoint assigned to fault i

The fraction's numerator represents the time it takes to run a fault-injection campaign without checkpointing. The denominator illustrates the time required by a fault-injection campaign which uses checkpointing. The value of S increases with the amount of simulation time saved through checkpointing. Hence, faults injected closer to the end of a simulation lead to shorter safety-verification campaigns.

Fault-injection campaigns, which are using SFI, distribute faults uniformly across a fault-injection simulation. For instance, consider a fault-injection campaign with 10 000 faults and a simulation of 2500 simulation time steps. Uniform

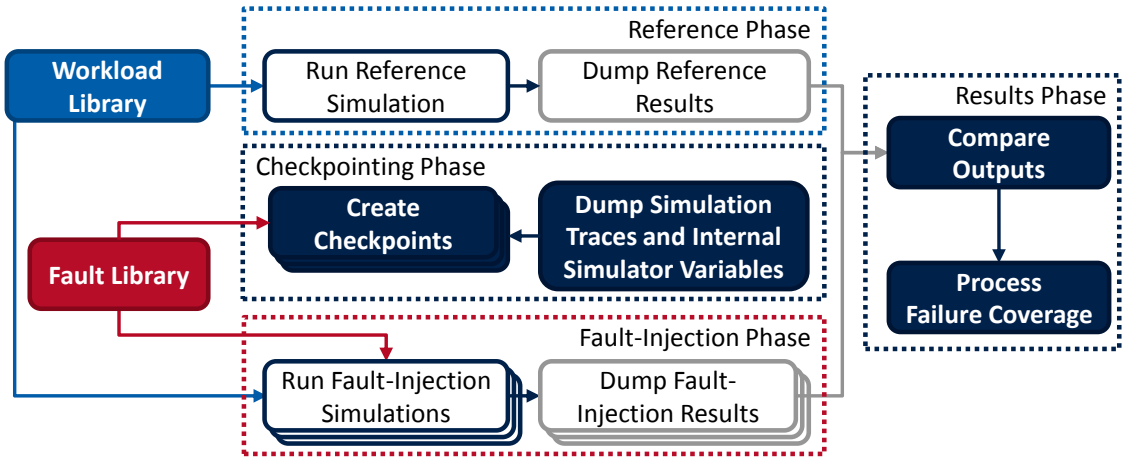


Figure 5.8.: SaVer's checkpointing-based fault-injection regression flow

fault distribution leads to injection of four faults at each simulation step but into different system components. Hence, N_{faults} can be written as a multiple of t_{sim}:

$$N_{faults} = K \cdot t_{sim}, K \in \mathcal{R} \tag{5.4}$$

where K represents the number of times each time step is used for fault injection.

After introducing Eq. 5.4 and expanding the sum, Eq. 5.3 becomes:

$$S = \frac{K \cdot t_{sim}^2}{K \cdot t_{sim}^2 - \sum_{i=1}^{N_{faults}} t_{chkpt_i}} \tag{5.5}$$

As shown in this thesis, fault-injection campaigns require complicated fault libraries. Nevertheless, faults can be injected into each simulation step (i.e., 1, 2, 3, ..., t_{sim}) using a uniform distribution. Also, faults injected into different parts of a system can share the same fault-injection time. In turn, such faults also share the same simulation checkpoint. Consequently, the sum term in Eq. 5.5 can be rewritten as the sum of an arithmetic series scaled by the K factor:

$$\sum_{i=1}^{N_{faults}} t_{chkpt_i} = \lfloor K \rfloor \cdot \frac{t_{sim} \cdot (t_{sim} + 1)}{2} + N_{faults} \bmod t_{sim} \tag{5.6}$$

where

$\lfloor K \rfloor$ = the floor of K
$N_{faults} \bmod t_{sim}$ = the remainder of dividing N_{faults} to t_{sim}

For instance, consider a fault-injection campaign of 17 faults injected into

simulations of five time steps. A uniformly distributed fault library requires 3 checkpoints for each simulation time step (i.e., $\lfloor K \rfloor = \lfloor 17/5 \rfloor = 3$). The remaining two faults (i.e., $N_{faults} \bmod t_{sim} = 17 \bmod 5 = 2$) are randomly attributed to any of the existing checkpoints (e.g., time steps 2 and 3).

By replacing the sum term into Eq. 5.5 and performing simplifications, S becomes:

$$S = \frac{2 \cdot K \cdot t_{sim}^2}{(K + \{K\}) \cdot t_{sim}^2 - \lfloor K \rfloor \cdot t_{sim} - 2 \cdot N_{faults} \bmod t_{sim}} \tag{5.7}$$

where $\{K\}$ is the fractional part of K, expressed as:

$$\{K\} = K - \lfloor K \rfloor \tag{5.8}$$

When simulation times are long, the speed-up factor becomes:

$$S' = \lim_{t_{sim} \to \infty} S = 2 \frac{K}{K + \{K\}} \tag{5.9}$$

On the one hand, for small fault libraries (i.e., $\{K\} = K$), there are too few faults for the checkpointing mechanism to have any benefit:

$$\lim_{K \to 0} S' = 1 \tag{5.10}$$

On the other hand, for huge fault libraries, the checkpointing mechanism achieves its maximum speed-up for uniformly distributed fault libraries:

$$\lim_{K \to \infty} S' = 2 \tag{5.11}$$

A 2x speedup factor is a significant simulation improvement when using simulation checkpointing because, until now, technological constraints (e.g., limited hard-disk space, manual checkpointing approaches) have prevented safety-verification flows from achieving any speedup whatsoever. The checkpointing mechanism introduced in this thesis makes this speedup factor possible.

5.7. Summary

SFI methods require thousands and even millions of simulations to provide confident results. Additionally, large safety-critical systems with complex workloads suffer from long simulation runs. Thus, fault-injection campaigns on

such systems are computationally intensive and require methods to reduce their execution time. To achieve this requirement, this chapter has introduced a safety-verification framework called SaVer, which automates the injection of faults into VPs. Furthermore, several optimization methods introduced in this chapter have been designed to reduce the simulation time and also the number of simulations of fault-injection campaigns.

One optimization method has been designed to group and remove redundant fault-injection locations, such as output ports, signals, and input ports. Since these locations exhibit the same fault-propagation paths and lead to the same fault-injection results, they can be reduced to a single fault-injection location. Thus, fewer fault-injection simulations are required to accurately verify the SoC.

Another optimization method has been designed to reduce the number of simulations in a fault-injection campaign. Compared to the previous method, this one analyzes the traces of a fault-free simulation and determines whether the injection of transient faults leads to fault propagation or not. As a result, faults, which are immediately masked by the system, are also removed from the fault-injection campaign.

The third optimization method speeds up the fault-injection campaign by allowing fault-injection simulations to run concurrently (in parallel). Thus, the speed of obtaining fault-injection results increases with the number of processor cores available to run simulations and respects Amdahl's Law [154].

The final method introduced in this chapter speeds up fault-injection simulations by using an optimized and automated simulation-checkpointing approach. This checkpointing mechanism achieves a speed-up factor of up to 2x, which is higher than any speedup achieved by previous approaches.

6. Experimental Results and Discussion

This chapter presents several HW architectures used to evaluate the methods described previously. Additionally, this chapter introduces the experimental setup used to perform fault-injection simulations and to measure fault effects, simulation outcomes, and performance results for the contributions introduced in this thesis. Finally, results are interpreted and the applicability of this thesis' contributions is presented.

6.1. Application Example

This section introduces several HW architectures used to selectively test and measure the techniques contributed in the previous two chapters. These architectures range from purely combinational adder circuits to complicated microprocessor cores. They do not contain any safety mechanisms.

6.1.1. Adder Architectures

This thesis experimented on various adder architectures such as carry-save adders and carry-lookahead adders, which range from full adders to 32-bit adders. The adders are modeled as SystemC/TLM-based VP and as equivalent gate-level net-lists. These models use the same interface: three inputs (i.e., two operands and one carry-in bit) and two outputs (i.e., the sum and a carry-out bit). The experiments use a VP with programmable operand bit-width for all adder architectures. The VP uses a TLM-target module with a blocking-transport method. Furthermore, the VP uses a dedicated TLM payload to model the adder interface. Consequently, the transport method simply performs the addition operation using the payload's attributes. The adders' gate-level net-lists are created using logic synthesis of each adder's RTL model using a commercial synthesis tool [155] and a generic technology file.

6.1.2. Microprocessor Cores

Besides the adder architectures, this thesis also experimented on three CPU cores: (i) microcontroller oc8051, (ii) AltOr32 (i.e., an alternative implementation of the Open RISC 1000 CPU), and (iii) NanoMIPS (i.e., a simplified in-house version of a MIPS CPU) [109], [113]. AltOr32 and oc8051 are thoroughly documented in [156] and in [157], respectively. Similar to the adder architectures, a VP and corresponding gate-level net-lists are used for the 8051 and AltOr32 CPUs. Gate-level net-lists are obtained by synthesizing the cores using a commercial tool [155] and a generic technology file. VPs are generated from gate-level net-lists using VERITAS and VERITAS++. NanoMIPS contains a 32-bit integer-based instruction set, hazard detection and forwarding units, and a five-stage pipeline. These features are mirrored within the gate-level net-list and VPs. The MIPS CPU is documented in [158].

NanoMIPS is modeled across multiple abstraction levels. It has an RTL model (Fig. 6.1) which contains a dedicated module for each pipeline stage. NanoMIPS also has a gate-level net-list generated from its RTL model using a commercial synthesis tool [155] and a generic technology file. Furthermore, NanoMIPS has three different VP models: a SystemC VP and two TLM VPs. While the SystemC VP closely resembles the RTL model's structure (Fig. 6.1), the TLM VPs are more abstract and are developed using the loosely-timed method (Fig. 6.2) and approximately-timed method (Fig. 6.3). Even here, the loosely-timed TLM model is more abstract than the approximately-time model. Both TLM models use a TLM-initiator module for the instruction-fetch pipeline stage. The other stages are TLM-interconnect modules. Finally, the instruction and data memories are TLM-target modules. Contrary to the RTL model, the write-back pipeline stage is integrated into the memory-access stage for model-simplification reasons. Each pipeline stage uses a specific set of TLM-payload extensions. The loosely-timed TLM model employs the blocking-transport method. The approximately-timed TLM model uses the non-blocking transport method. Besides NanoMIPS' manually implemented SystemC and TLM VPs, VERITAS was also applied to combinational blocks of NanoMIPS' gate-level net-list to transform them into TLM models. The resulting code augments NanoMIPS' TLM models (e.g., adders, look-up tables, shifters, branching units). Finally, VERITAS++ is used on the gate-level net-list to generate a complete gate-level-accurate SystemC/C++ version of NanoMIPS.

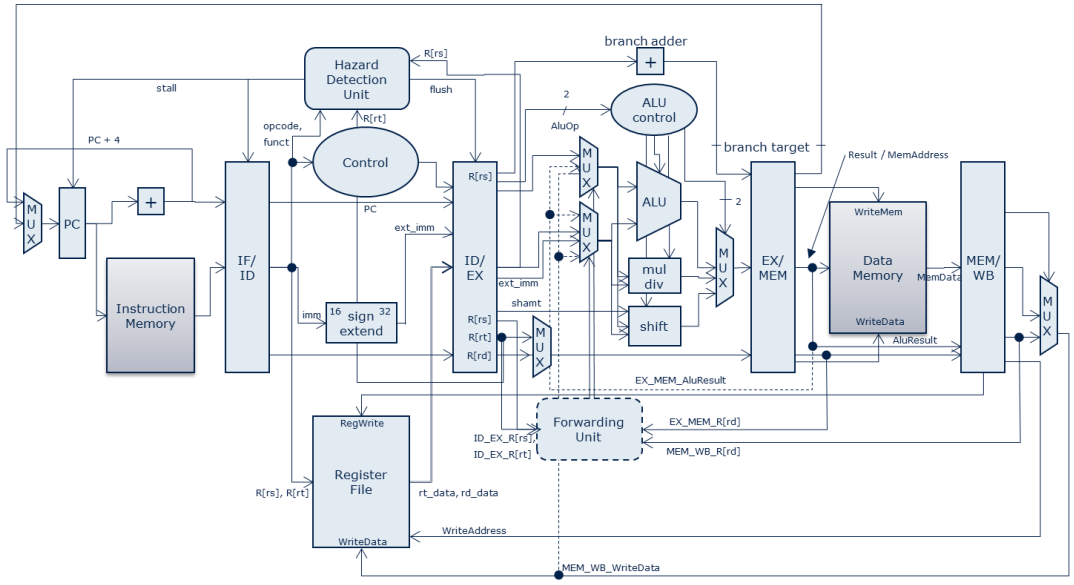


Figure 6.1.: Block diagram of NanoMIPS' SystemC, RTL, and gate-level model

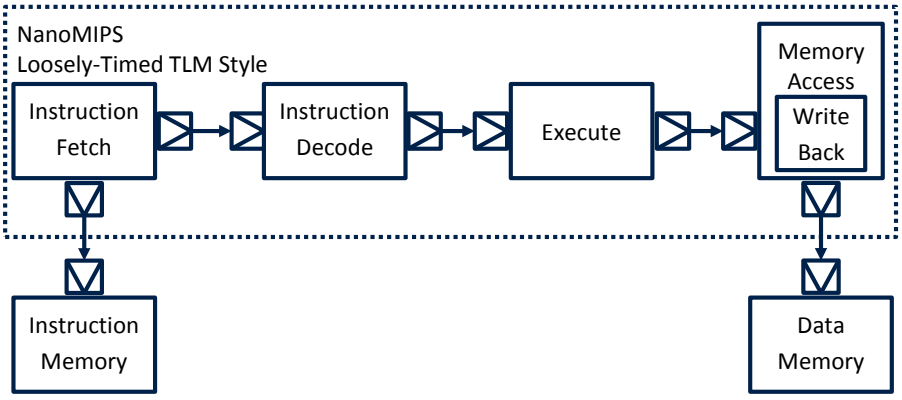


Figure 6.2.: Block diagram of NanoMIPS' loosely-time TLM model

6.2. Experimental Setup

The studies presented in this thesis inject faults randomly using Monte-Carlo simulation. Each test injects only one fault into internal system states (e.g., signals, buses, register, variables) and outputs of system blocks. These studies measure each HW model's base fault tolerance. The experiments on adder architectures inject random values into each operand. Additionally, they exhaustively test all adder architectures up to 16 bits in size (i.e., half, full, nibble, 8-bit, and 16-bit adders). The experiments on microprocessor cores use a broad range of firmware tests as input stimuli for the (e.g., ALU tests, interrupt-controller

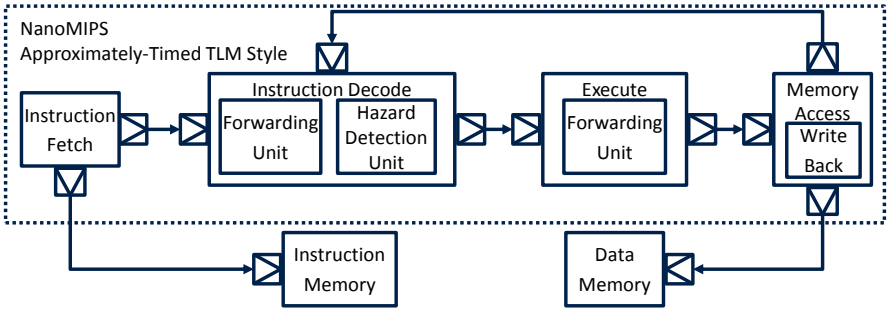


Figure 6.3.: Block diagram of NanoMIPS' approximately-time TLM model

tests, memory tests). The experiments execute the SystemC/TLM VPs with the SystemC reference simulator. They run the RTL models and gate-level net-lists with commercial simulators [93]–[95].

The experiment results have been used to calculate and compare several ratios such as speedup, correlation, simulation overhead, and others. These ratios have been averaged using the harmonic mean because it is more stable against extreme values than the geometric mean. The arithmetic mean has been excluded because it can lead to wrong results when comparing ratios [159].

All simulations have been performed on 64-bit computers with an Intel® Xeon® E5 CPU @3.00 GHz, L3 cache 25600 kB, and 264 GB RAM. Compilation and elaboration times of the simulated models are negligible and thus have been excluded from the results section. The simulation time of each case study has been measured with the UNIX *time* command.

6.3. Quantitative Analysis of Fault-Matching Points

This section presents a new approach to quantify the correlation factor of VPs and gate-level net-lists for several adder architectures and the NanoMIPS CPU core. It does this by quantifying the number of fault-matching points (i.e., internal signals and output ports) available within the VPs and gate-level net-lists:

$$Correlation = \frac{N_{VP}}{N_{GL}} \tag{6.1}$$

where

N_{VP} = number of fault-matching points on the VP
N_{GL} = number of fault-matching points on the gate-level net-list

Additionally, the approach calculates the augmentation factor:

$$Augmentation = \frac{1}{Correlation} \tag{6.2}$$

Finally, the approach augments the VPs with gate-level information using VERITAS and VERITAS++. This step enriches the original VPs with all fault-matching points present on the gate-level net-lists. Thus, the VP's correlation factor is increased to 100%.

In the case of the adder architectures, the correlation factor depends on the adder's size: the larger the adder, the more logic gates it contains, and thus, the lower the correlation (Table 6.1). Based on the performed experiments, the highest correlation factor did not exceed 34% in the case of a full adder while the average value was below 13%. Hence, adders larger than 8 bits exhibit a correlation below the recorded harmonic mean. Additionally, some adder architectures, such as the carry-lookahead adder, further reduce the correlation factor because of the high gate-level complexity compared to the VP. Nevertheless, all of these poor-correlating adders benefit significantly from augmentation using VERITAS and VERITAS++. As already mentioned, the augmentation factor is the inverse of the correlation factor. In other words, the lower the correlation factor of a VP, the more that VP benefits from augmentation. Thus, the VPs enriched with VERITAS and VERITAS++ were augmented, on average, approximately sixfold. The 32-bit carry-lookahead adder recorded the highest augmentation (i.e., one order of magnitude).

Table 6.1.: Adders–Correlation factor across the VP and gate abstraction levels

A	B	C	D = B/C	E = C/B
Model Name	Injectable Fault Locations (#)		Correlation	Augmentation
	TLM Model	Gate-Level Net-List	Factor (%)	Factor
full_adder	2	6	33.33	3.00x
adder_2bit	3	9	33.33	3.00x
nibble_adder	5	31	16.13	6.20x
adder_4bit	5	35	14.29	7.00x
addsub_8bit	9	69	13.04	7.67x
adder_8bit	9	77	11.69	8.56x
adder_16bit	17	163	10.43	9.59x
adder_32bit	33	339	9.73	10.27x
carry_lookahead_32bit	33	523	6.31	15.85x
Harmonic Mean			12.65	6.07x

Measurements performed on the NanoMIPS core provide similar results to those obtained on the adder modules (Table 6.2). Multiple combinational TLM blocks correlate poorly with their corresponding gate-level net-lists. Similar to the adder architectures, the augmentation of NanoMIPS' combinational blocks was, on average, almost sixfold. Even here, the approach recorded a VP augmentation greater than one order of magnitude for some blocks such as id_control, hazard_detection, and logic shift.

Table 6.2.: NanoMIPS–Correlation factor across the VP and gate abstraction levels

A	B	C	D = 100*B/C	E = C/B
Model Name	Injectable Fault Locations (#)		Correlation	Augmentation
	TLM Model	Gate-Level Net-List	Factor (%)	Factor
write_back	38	82	46.34	2.16x
jump	3	10	30.00	3.33x
subtractor	33	174	18.97	5.27x
adder	33	209	15.79	6.33x
logic_unit	32	266	12.03	8.31x
ex_forwarding	4	38	10.53	9.50x
id_control	11	116	9.48	10.55x
hazard_detection	4	73	5.48	18.25x
shift	32	931	3.44	29.09x
Harmonic Mean			9.70	5.92x

These results confirm the expectations from Chapter 4.4: user-developed VPs and gate-level net-lists correlate poorly because VPs lack sufficient fault-matching points. Furthermore, these results indicate the benefits of VERITAS and VERITAS++ over traditional VP-based safety-verification approaches. Thus, the accuracy of safety verification on VPs can be significantly improved by augmenting VPs with gate-level information. VERITAS and VERITAS++ increase the correlation factor to 100%, add missing fault-matching points to VPs, enrich the models with correct fault-propagation paths, and, by extension, provide the VPs with more fault-masking effects.

6.4. Qualitative and Quantitative Analysis of Permanent-Fault Effects

This section documents the approach to qualitatively and quantitatively analyze the effects of permanent faults (i.e., stuck-at-0 and stuck-at-1) on the adder

architectures and combinational blocks of the NanoMIPS core. This approach uses the Monte-Carlo method to cover a broad spectrum of input stimuli and fault-injection patterns.

After simulating 10 000 samples for each fault model, the approach observed all possible single-bit failures in the adder's *sum* output (Fig. 6.4). These findings are independent of adder size and fault type (i.e., stuck-at-0 and stuck-at-1), type of HW models used (i.e., TLM-based VPs and gate-level net-lists), and fault-injection locations (i.e., at the input-interface level or internal). However, the Monte-Carlo approach is not as efficient when considering corner cases (i.e., less probable outcomes). On the one hand, random stuck-at-1 injection into the adders has successfully led to failure observation (i.e., failure coverage) in the *carry-out* output of all adder sizes (the bars below the 50% coverage margin in Fig. 6.5). On the other hand, stuck-at-0 faults have only registered these effects for 8-bit and 16-bit adders. In the case of the 32-bit adders, the approach did not register a carry-out failure even after 10 000 000 simulations. This result can be visualized as the missing bars above the 50% coverage margin. Consequently, directed tests are necessary to cover the *carry-out* corner case and to reduce the total verification time of the 32-bit adders.

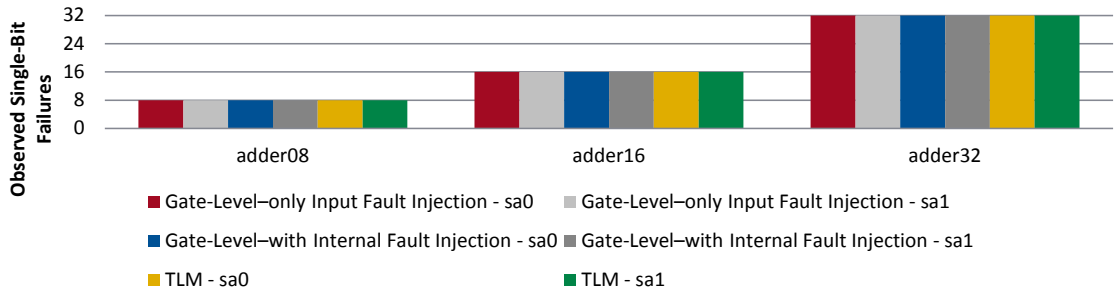


Figure 6.4.: Observed number of faulted bit positions in the sum output of an 8, 16, and 32-bit adder after 10 000 samples

Next, the approach quantifies the differences of injecting faults into gate-level and TLM models by comparing the size of the observed multi-bit failures after injecting stuck-at-1 (Fig. 6.6) and stuck-at-0 (Fig. 6.7) faults. An n-bit adder offers a maximum multi-bit failure of n+1 bits given by its n-bit *sum* output and 1-bit *carry-out* output. This study used the same input patterns for both abstraction levels. First, fault injection into (i) inputs of the adders' gate-level net-list, (ii) inputs and internal signals of these gate-level net-lists, and (iii) inputs of their corresponding TLM models offered similar simulation results. Second, to observe multi-bit failures, the necessary average number of random samples

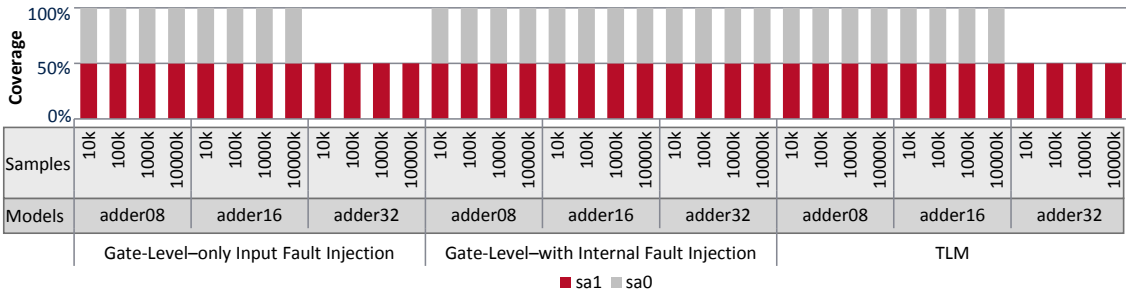


Figure 6.5.: Coverage of carry-out failures per fault type observed for 8, 16, and 32-bit adder

depends on the adder's size. This number becomes unmanageable as this size increases. For the 8-bit adder, 10 000 samples generated all possible multi-bit failures (i.e., up to 9 bits) for both permanent fault models. However, stuck-at-0 injection has offered better results than stuck-at-1 insertion for the 16-bit adder. In this case, stuck-at-0 faults needed only 1 000 000 samples to cover all multi-bit failures (i.e., up to 17 bits), while stuck-at-1 faults barely covered 16 bits after 10 000 000 samples. Similarly, 10 000 000 samples only covered multi-bit failures up to 21 bits for the 32-bit adder, regardless of fault model. Directed testing is necessary to include these missing scenarios and to perform exhaustive safety verification on the adders.

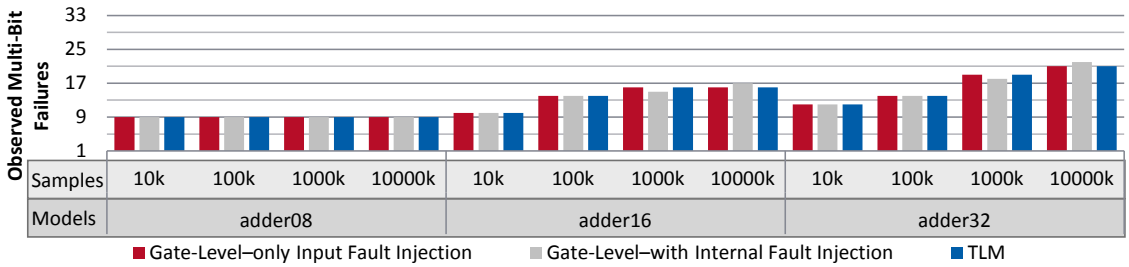


Figure 6.6.: Maximum multi-bit failure observed after injecting stuck-at-1 faults in adders of variable sizes

This approach calculates the adder's failure probability to quantify the number of random samples needed, on average, by an n-bit adder to cause a maximum multi-bit failure. The failure probability is calculated as a function of the adder's size. In this case, the approach considers fault injection only into the model's inputs. The total number of input stimuli subject to fault injection give the total number of all outcomes (AO). This number of stimuli exhaustively covers all input bit configurations of an n-bit adder. Given the adder's size (i.e., n-bits)

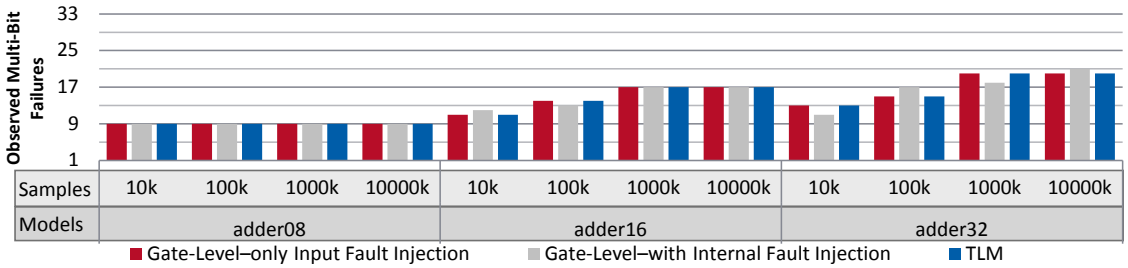


Figure 6.7.: Maximum multi-bit failure observed after injecting stuck-at-0 faults in adders of variable sizes

and its inputs (i.e., two n-bit inputs and one carry-in bit), the number of input stimuli is:

$$N_{stimuli} = 2^{2n+1} \tag{6.3}$$

where n is the adder's size in bits. Additionally, any of the adder's 2n+1 input bits supports stuck-at fault injection, regardless of the given input pattern. Thus, AO becomes:

$$AO = (2n+1) \cdot 2^{2n+1} \tag{6.4}$$

Next, the approach derives the number of samples necessary to cause an n+1-bit failure in an n-bit adder. Two cases exist which lead to the observation of maximal multi-bit failures. First, consider a full adder with the input pattern a=0, b=0, and carry-in=1 which leads to the outputs carry-out=0 and s=1. A stuck-at-1 fault injected into either a or b causes a 2-bit failure by flipping the output bits (i.e., carry-out=1 and s=0). Second, consider the input pattern a=0, b=1, and carry-in=1 which leads to the outputs carry-out=1 and s=0. A stuck-at-0 fault injected into either b or carry-in causes a 2-bit failure by flipping the output bits (i.e., carry-out=0 and s=1). As a result, the approach determined the number of desired outcomes (DO) for any particular stuck-at fault experimentally and generalized it as:

$$DO = 2(2^n + 2^{\lceil \frac{n-1}{n} \rceil}) \tag{6.5}$$

where

$\lceil K \rceil$ = the ceiling of number K

Thus, the probability of causing an n+1-bit failure by randomly generating

input patterns and injecting a particular stuck-at fault becomes:

$$P = \frac{DO}{AO} \tag{6.6}$$

Finally, the average number of simulations with random stimuli required to cause a maximum MBF is:

$$N_{avg} = \frac{1}{P} \tag{6.7}$$

The approach applied Eq. 6.7 on various adder sizes to illustrate the average number of Monte-Carlo simulations (Table 6.3). The probability of obtaining a maximum multi-bit failure for a 32-bit adder is intrinsically small (i.e., about $3.58 \cdot 10^{-10}$%). These results also validate the experimental results. The 10 000 000 samples used in the Monte-Carlo simulation were not sufficient to obtain a maximum multi-bit failure. Additionally, while 4318 simulation for an 8-bit adder are still feasible, current technological limitations cannot simulate over two million samples for 16-bit adders in a reasonable amount of time. Therefore, directed testing becomes a powerful method in overcoming this limitation.

Table 6.3.: Average number of simulations required to cause a maximum multi-bit failure in several adders

Adder Size	Average Number of Simulations
1	4
8	4318
16	2 162 622
32	279 172 874 110

For NanoMIPS, 10 000 samples have been sufficient to cause at least one single-bit failure in all outputs of NanoMIPS' combinational sub-blocks (Fig. 6.8). Besides this, the results of multi-bit fault injection coincide with those from the previous case study (Fig. 6.9). As in the case of the adders, a multi-bit failure's size increases with the number of output bits of a sub-block. For instance, consider the write_back block, which contains 32 output bits. A fault-injection campaign of 1 000 000 samples only led to one 29-bit failure. Consequently, directed tests are necessary to cover the remaining multi-bit failures. These results apply to both the TLM models and gate-level net-list and are fault-model independent.

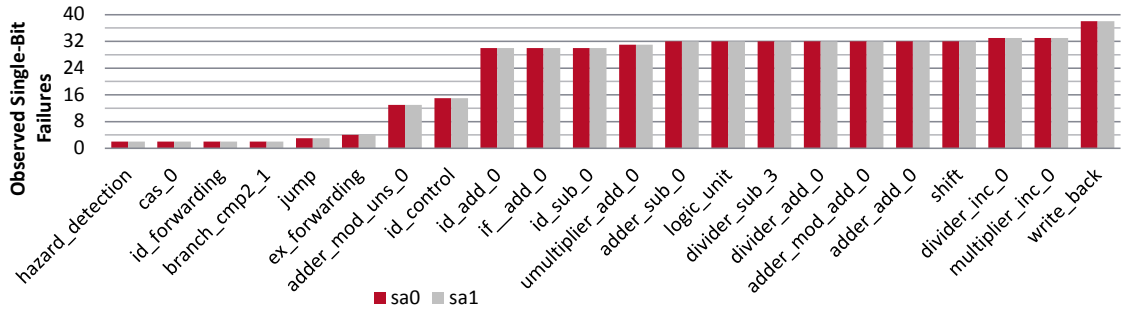


Figure 6.8.: Single-bit failures observed for various MIPS CPU gate-level sub-blocks after injecting permanent faults into 10 000 random samples

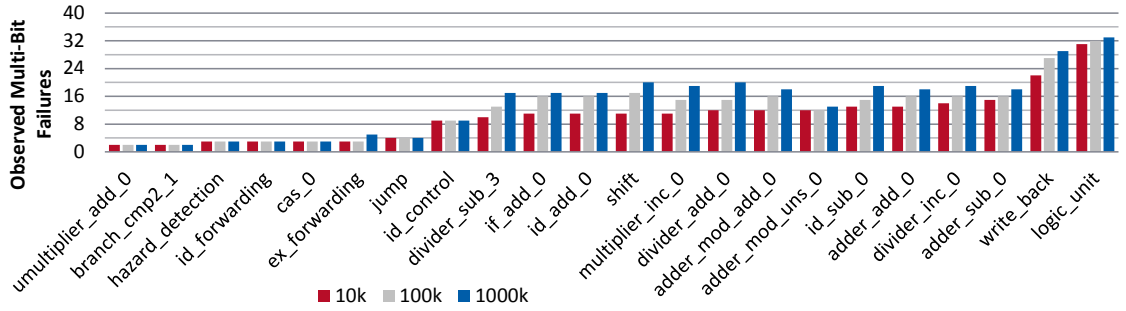


Figure 6.9.: Multi-bit failures observed for several MIPS CPU gate-level sub-blocks after injecting stuck-at-1 faults

6.5. Simulation Performance of Fault-Injection Methods

This section presents different case studies which measured the average computational overhead of SCFIT, injectable TLM sockets, and FIOs.

6.5.1. SCFIT

This test case measures SCFIT's simulation overhead on the manually developed SystemC models of NanoMIPS (Table 6.4). The test case grouped this measurement into one fault-free simulation (injection of 0 faults), followed by simulation with up to three faults. Column *B* represents the amount of time required to run a NanoMIPS simulation without SCFIT. Based on these measurements, SCFIT's usage of Python and GDB introduces a static simulation overhead of 1-2%. Furthermore, SCFIT introduces a simulation overhead of 10-11% when injecting a single fault into the SystemC VP. However, when injecting two or

more faults, the simulation overhead more than doubles because SCFIT exceeds the four HW breakpoints available on Intel x86_64 CPUs. In this case, GDB creates software breakpoints which dramatically reduce the performance of the fault-injection simulation. Nevertheless, single-bit fault injection is currently sufficient for safety verification [5], which makes SCFIT an efficient tool. SCFIT provides a useful measure for single-bit and multi-bit (e.g., two or three bits) fault-injection campaigns, especially for precompiled modules, which do not allow model changes.

Table 6.4.: Simulation overhead measured for SCFIT

A	B	C	D = 100*(C/B-1)
Faults Injected per Simulation	Reference Simulation Time (s)	Fault-Injection Simulation Time (s)	Simulation Overhead (%)
0	3.71	3.771	1.62
1	3.71	4.146	10.52
2	3.71	4.713	21.28
3	3.71	4.899	24.27
Harmonic Mean			4.99

6.5.2. Injectable TLM Sockets

This study measures the simulation overhead of injectable TLM sockets on manually implemented TLM models of the NanoMIPS core (Table 6.5). This study considered the loosely-timed TLM blocking-transport method. Nevertheless, the injectable TLM sockets support fault injection using the TLM non-blocking-transport method. Similar to SCFIT, this study measures the tools' performance by running fault-injection simulations with up to three fault injections, respectively. Next, the study compared the simulation results to a reference simulation on equivalent TLM models which did not use injectable TLM sockets. These results show a greater static overhead during fault-free simulations than SCFIT. Nevertheless, the injectable TLM sockets offer a clear improvement in simulation performance when injecting single and even multiple faults. In this case, the simulation overhead is less than half compared to SCFIT. The first reason for this improvement is because injectable TLM sockets do not require any breakpoints. Thus, TLM sockets support significantly more fault injections into a system than SCFIT for the same performance penalty. The second reason is that fault-injection

simulations use compiled injectable TLM sockets, which provide faster results compared to SCFIT's interpreted Python code.

Table 6.5.: Simulation overhead measured for injectable TLM sockets

A	B	C	D = 100*(C/B-1)
Faults Injected per Simulation	Reference Simulation Time (ms)	Fault-Injection Simulation Time (ms)	Simulation Overhead (%)
0	248.538	253.706	2.04
1	248.538	258.934	4.02
2	248.538	272.149	8.68
3	248.538	274.137	9.34
Harmonic Mean			4.16

6.5.3. Fault-Injection Objects

This case study applies FIOs on the SystemC models generated for NanoMIPS, oc8051, and AltOr32 using VERITAS and VERITAS++ and also on the manually-developed SystemC models of NanoMIPS. This study benchmarks FIOs against C++ Boolean variables after compiling the code using the GNU G++ compiler and the *O3* optimization flag. Additionally, it declares all HW registers as wires present within each core as FIOs. Each CPU core uses firmware tests which executed 2^{32} register accesses (i.e., read and write) over 100 simulations. Results show an average 2% simulation overhead compared to Boolean variables. Simulations using FIOs required, on average, 1216.798 ms, compared to the simulation with C++ boolean variables which needed on average 1240.866 ms. Furthermore, the study recorded a negligible simulation overhead when injecting multiple faults per simulation. Hence, FIOs are more efficient than SCFIT and injectable TLM sockets.

6.6. Performance Measurements of Checkpointing Mechanism

The following test cases measure the performance of the checkpoint-restore mechanism presented in Chapter 5.6. Measurements have been made on the

oc8051, AltOr32, and NanoMIPS cores by analyzing the following three categories: hard-disk-space requirements, generation time, and safety-verification speed-up.

6.6.1. Requirements of Hard-Disk Space

The test case measures the hard-disk-space requirements for SaVer's checkpointing mechanism in two situations: for one checkpoint and 1000 checkpoints. Additionally, this test case compares the results to three commercial HW simulators with checkpointing capabilities. The first two simulators are event-driven. The third one is a compiled-code simulator. Even though 1000 seems like an arbitrary number, current computational constraints (i.e., insufficient hard-disk space) limit this value. However, the resulting information far exceeds what is typically presented in similar scientific studies and adds an extra benefit to this thesis.

When generating a single checkpoint, SaVer outperforms all three commercial tools, on average, by one to two orders of magnitude (Table 6.6, columns F, H, and J). The main reason for this difference is SaVer's optimized checkpointing method. SaVer only saves a VCD file for the analyzed HW model and the internal states of Verilator's simulator (i.e., only five variables for each test CPU core). Conversely, checkpoints saved by commercial tools contain significantly more information (e.g., waveform data, annotated time delays, scheduled events, simulator configuration). These surplus data ultimately increase the hard-disk size of each commercial tool's snapshot.

Table 6.6.: Hard-disk-space requirements for one checkpoint

A	B	C	D = C/B	E	F = E/B	G	H = G/B
CPU	Verilator (MB)	Vendor I (MB)	Reduction Factor	Vendor II (MB)	Reduction Factor	Vendor III (MB)	Reduction Factor
oc8051	0.136	4.1	30.15x	6.5	47.79x	33.3	244.85x
NanoMIPS	0.585	15.0	25.64x	20.0	34.19x	37.3	63.76x
AltOr32	0.205	5.6	27.32x	8.3	40.49x	33.8	164.88x
Harmonic Mean			27.58x		40.07x		116.13x

When generating multiple checkpoints, SaVer outperforms the commercial tools by three to four orders of magnitude. SaVer only requires 0.5 MB to 2.5 MB to generate 1000 checkpoints whereas commercial tools require several GBs (Table 6.7, columns D, F, and H). SaVer achieves this by splitting each

checkpoint file into a header file and multiple data files. The header file contains all checkpointed signal names. Each data file contains the signals' values for the given checkpoint's simulation time stamp. As a result, SaVer generates a single data-heavy header file and several lightweight data files.

Table 6.7.: Hard-disk-space requirements for 1000 checkpoints

A	B	C	D = C/B	E	F = E/B	G	H = G/B
CPU	Verilator (MB)	Vendor I (GB)	Reduction Factor	Vendor II (GB)	Reduction Factor	Vendor III (GB)	Reduction Factor
oc8051	0.76	30.18	39.86 Kx	6.50	8.60 Kx	47.84	63.20 Kx
NanoMIPS	2.30	110.40	48.00 Kx	20.02	8.70 Kx	147.21	64.00 Kx
AltOr32	0.71	41.22	57.80 Kx	8.31	11.65 Kx	61.09	85.68 Kx
Harmonic Mean			47.46 Kx		9.46 Kx		69.58 Kx

6.6.2. Generation Time of Checkpoints

Checkpoints not only need a lot of hard-disk space, but they also require a long time to generate. This study measures SaVer's checkpoint generation time for 1000 checkpoints and compared it to three commercial tools. SaVer's checkpointing method is, on average, 5 to 11 times faster than the commercial tools (Table 6.8, columns D, F, and H). This speed can be further improved by replacing the Python-based checkpoint generator with a C/C++ implementation and by using a more efficient format for the checkpoint's contents than ASCII. The latter would also further decrease the size of checkpoints.

Table 6.8.: Generation time for 1000 checkpoints

A	B	C	D = C/B	E	F = E/B	G	H = G/B
CPU	Verilator (s)	Vendor I (s)	Reduction Factor	Vendor II (s)	Reduction Factor	Vendor III (s)	Reduction Factor
oc8051	0.41	3.61	8.80x	2.09	5.10x	6.47	15.78x
NanoMIPS	1.06	9.91	9.35x	4.13	3.90x	7.51	7.08x
AltOr32	0.47	5.51	11.72x	3.31	7.04x	6.96	14.81x
Harmonic Mean			9.81x		5.04x		11.03x

6.7. Reduction of Fault-Verification Space

This section presents the results of performance analysis of the methods introduced in this thesis to reduce the fault-verification space of safety-critical SoCs.

6.7.1. Spatial Fault Pruning

This approach analyzes the effectiveness spatial-fault pruning as introduced by VERITAS++. After running this approach on three cores, the two methods classified around half of the signals present on the gate-level net-list as redundant and removed them from the safety verification (Table 6.9). This result corresponds to halving the fault-injection-locations attribute from the three cores' fault-verification spaces and overall minimizing each core's fault library.

Table 6.9.: VERITAS++–Spatial fault pruning

A	B	C	D = B/C
CPU	Gate-Level Signals	C++ Signals	Optimization Factor
oc8051	8646	3903	2.22x
AltOr32	12 051	6166	1.95x
NanoMIPS	36 570	14 178	2.58x
Harmonic Mean			2.22x

6.7.2. Temporal Fault Pruning

This study measures the performance gain obtained from applying SaVer's temporal-fault-pruning feature on SystemC models of NanoMIPS (Table 6.10). SaVer implements temporal fault pruning by analyzing the simulation traces of registers and signals (see Chapter 5.4.2). First, this study conducts a fault-injection campaign to determine the effectiveness of classic Monte-Carlo-based safety verification. Next, it applies temporal fault pruning to optimize the safety verification for transient faults by reducing the number of fault-injection locations and fault-injection simulations. Finally, this study quantifies the benefit of temporal fault pruning by calculating the reduction factor which this feature introduces. After injecting over one million transient faults (one fault per simulation) into NanoMIPS' 88 SystemC registers using the Monte-Carlo approach, only 13.80% of these faults led to failures while the system masked

the remaining ones. However, fault pruning on NanoMIPS classified 31 registers as safe because they could not lead to failures from injection of transient faults, which led to a 1.84-fold initial reduction. Furthermore, lifetime analysis on the remaining 57 registers generated only 33 275 simulations valid for transient-fault injection. This result led to 33.88-fold fewer fault-injection simulations compared to the Monte-Carlo approach without pruning. Finally, after injecting a transient fault into each of the resulting simulations, over 88% of the faults led to failures. SaVer's temporal-fault-pruning approach is over 70% more efficient than Monte-Carlo simulation without pruning.

Table 6.10.: SaVer–Temporal fault pruning

A	B	C	D	E	F	G	H=B/E	I=C/F
Model	Fault Injection at Random Simulation Time			Fault Injection into Valid Time Intervals			Reduction Factor	
	Accessed Signals	Number of Simulations	Effectiveness Percentage	Accessed Signals	Number of Simulations	Effectiveness Percentage	Accessed Signals	Number of Simulations
NanoMIPS	88	1 127 280	13.80%	57	33 275	88.62%	1.54x	33.88x

6.8. Speed-Up of Fault-Effect Analysis

This section presents several test cases which quantify the accuracy and speed-up improvements of VERITAS and VERITAS++ when performing VP-based safety verification. This section also reports the benefits of three safety-verification optimization techniques: (i) spatial and (ii) temporal fault pruning and (iii) simulation checkpointing.

6.8.1. VERITAS and VERITAS++

This study measures VERITAS' performance on adder modules and combinational blocks of NanoMIPS (e.g., ALU operations, instruction decode and write-back block) [110], [111]. During this study, VERITAS adds gate-level-specific information to the TLM models (see Chapter 6.3). As expected, the TLM-based adder models augmented with gate-level information introduce a noticeable simulation overhead. These models are, on average, four times slower than the original TLM models (i.e., without gate-level augmentation) (Table 6.11). This simulation overhead increases with the adder's size. Additionally, the

adder's architecture influences the simulation complexity (i.e., a 32-bit carry-lookahead adder has a more complicated structure than a simple 32-bit carry adder).

Table 6.11.: Adders–Simulation overhead introduced by VERITAS

A	B	C	D = B/C
Model Name	Simulation Time (s)		Slow-Down Factor
	Augmented TLM Model	Original TLM Model	
full_adder	0.041	0.040	1.03x
adder_2bit	0.097	0.087	1.11x
nibble_adder	0.260	0.122	2.13x
adder_4bit	0.310	0.122	2.54x
addsub_8bit	0.690	0.194	3.57x
adder_8bit	1.600	0.194	8.27x
adder_16bit	3.470	0.405	8.58x
adder_32bit	13.550	1.142	11.87x
carry_lookahead_32bit	16.040	1.142	14.05x
Harmonic Mean			2.64x

Even though the augmented TLM models have slower execution than the original TLM models, they run two to five times faster than the original gate-level net-lists, as indicated by the bottom bars in Fig. 6.10. These bars are below the 50% line for each adder because the augmented TLM models require less execution time than the gate-level net-list. Furthermore, this speedup improves with adder complexity. Thus, the bigger and more complicated the adder architecture is, the higher the simulation speed-up becomes. For instance, the speed-up of the carry-lookahead adder is six times higher than the speed-up of the 2-bit adder.

Augmented TLM models have lower performance than pure TLM models but much higher than gate-level net-lists. Thus, it is necessary to determine the benefit of applying VERITAS on gate-level net-lists. First, fault injection into VPs alone can lead to observation of pseudo-faults (see Chapter 4.3). Therefore, safety experts must perform fault-injection simulations on gate-level net-lists to validate the results obtained on VPs. This process leads to independent fault-injection campaigns on each abstraction level. However, by merely performing fault injection into the augmented TLM models, safety experts can avoid such subsequent fault-injection campaigns. Second, it is worth comparing the trade-off between simulation performance and the improved correlation of VPs and gate-level net-lists (Table 6.12). This trade-off factor is determined by first multiplying

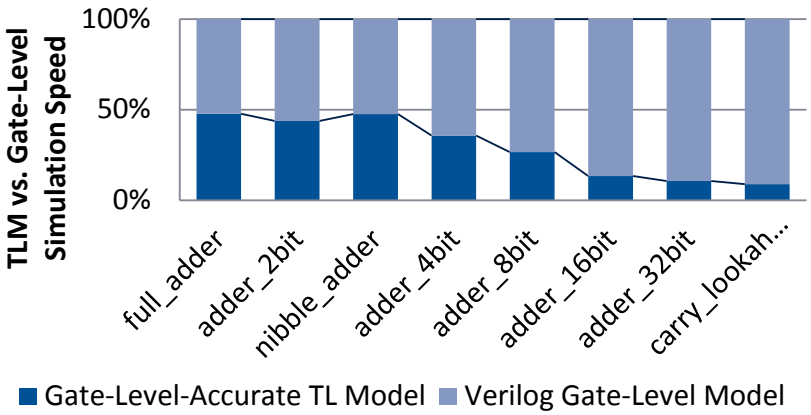


Figure 6.10.: Simulation-speed comparison between each gate-level and gate-level-accurate TLM

the VP augmentation factor (i.e., increase in simulation complexity) with the VP speed-up factor over gate-level simulation and then by dividing the VP slow-down factor over pure TLM simulation. If the resulting trade-off factor is higher than one, the augmentation of TLM models with gate-level information improves both the speed and accuracy of the safety verification. If the trade-off is less than one, safety experts should avoid augmenting their VPs.

Table 6.12.: Adders–Simulation performance vs improved fault-injection correlation

A	B	C	D	E = B*C/D
Model Name	Augmentation of TLM (Factor)	Speed-Up vs. Gate Level (Factor)	Slow-Down vs. TLM (Factor)	Trade-Off (Factor)
full_adder	3.00x	1.09x	1.03x	3.20x
adder_2bit	3.00x	1.28x	1.11x	3.48x
nibble_adder	6.20x	1.10x	2.13x	3.20x
adder_4bit	7.00x	1.80x	2.54x	4.96x
addsub_8bit	7.67x	2.76x	3.57x	5.93x
adder_8bit	8.56x	6.43x	8.27x	6.66x
adder_16bit	9.59x	6.46x	8.58x	7.23x
adder_32bit	10.27x	8.41x	11.87x	7.28x
carry_lookahead_32bit	15.85x	10.29x	14.05x	11.61x
Harmonic Mean	6.07x	2.23x	2.64x	5.01x

The adder modules have a trade-off factor between about three and 12. Thus, VERITAS proves to be a useful tool for improving the accuracy and speed of

safety verification on combinational circuits. NanoMIPS offers similar measurement results. This study compares augmented TLM models to NanoMIPS's loosely-timed TLM-style models and equivalent NanoMIPS RTL models (Table 6.13). Furthermore, it replaces individual combinational TLM modules with augmented ones. Then, it compares the resulting system modules to the original RTL and TLM models. Similar to the adder modules, simulation performance depends on the complexity of augmented TLM models. The more complicated the model is, the greater the simulation overhead compared to the pure TLM model. However, the speed-up over RTL decreases with the increase in complexity. Finally, this study calculates the trade-off between simulation performance and improved fault-injection correlation (Table 6.14). All combinational blocks contain trade-off factors greater than 1 by 1-2 orders of magnitude (i.e., up to 340-fold for the jump block). The only exception is the shift block which still has a trade-off value greater than 2. These results further strengthen VERITAS' effectiveness.

Table 6.13.: NanoMIPS–Simulation performance introduced by VERITAS

A	B	C	D	E = B/C	F = D/B
Model Name	Simulation Time (s)			Simulation Slow-Down Factor	Abstraction Speed-Up Factor
	Augmented TLM Model	Original TLM Model	RTL Model		
write_back	0.380	0.24	70.831	1.58x	186.40x
jump	0.408	0.24	70.831	1.70x	173.56x
subtractor	2.164	0.24	70.831	9.02x	32.73x
adder	2.593	0.24	70.831	10.80x	27.32x
logic_unit	2.761	0.24	70.831	11.50x	25.65x
ex_forwarding	1.192	0.24	70.831	4.97x	59.40x
id_control	1.914	0.24	70.831	7.98x	43.99x
hazard_detection	1.610	0.24	70.831	6.71x	37.01x
shift	13.564	0.24	70.831	56.51x	5.22x
Harmonic Mean				4.49x	23.97x

A different study measures the performance and effectiveness of VERITAS++ on three CPU cores and compares these results with the results obtained from three commercial simulators (Table 6.15). The simulators from Vendors I and II are event-driven. The one from Vendor III is a compiled-code simulator (similar to Verilator's). By using Verilator, VERITAS++ outperforms all three commercial tools on all accounts. The increase in simulation performance ranges, on average, from 1.6-fold to over 5.5-fold. Compared to gate-level

Table 6.14.: NanoMIPS–Simulation speed vs improved fault-injection correlation

A	B	C	D	E = B*C/D
Model Name	Augmentation of TLM (Factor)	Speed-Up vs. RTL (Factor)	Slow-Down vs. TLM (Factor)	Trade-Off (Factor)
write_back	2.16x	186.40x	1.58x	254.58x
jump	3.33x	173.56x	1.70x	340.31x
subtractor	5.27x	32.73x	9.02x	19.13x
adder	6.33x	27.32x	10.80x	16.02x
logic_unit	8.31x	25.65x	11.50x	18.54x
ex_forwarding	9.50x	59.40x	4.97x	113.54x
id_control	10.55x	43.99x	7.98x	58.13x
hazard_detection	18.25x	37.01x	6.71x	100.66x
shift	29.09x	5.22x	56.51x	2.69x
Harmonic Mean	5.92x	23.97x	4.49x	15.42x

simulation speed, the models generated using VERITAS++ are approximately three orders of magnitude faster. This analysis shows that VERITAS++ is an excellent complementary approach to VERITAS as it is usable on sequential HW elements as well (e.g., flip-flops, registers).

Table 6.15.: Simulation performance of VERITAS++

A	B	C	D = C/B	E	F = E/B	G	H = G/B
CPU	VERITAS++ (s)	Vendor I (s)	Speed-Up Factor	Vendor II (s)	Speed-Up Factor	Vendor III (s)	Speed-Up Factor
oc8051	466.01	755.09	1.62x	755.09	1.62x	3031.3	6.50x
NanoMIPS	1216.80	1840.35	1.51x	1950.31	1.60x	4693.5	3.86x
AltOr32	511.58	978.63	1.91x	955.60	1.87x	4438.4	8.68x
Harmonic Mean			1.67x		1.69x		5.68x

6.8.2. Checkpointing Mechanism

SaVer's automated checkpointing method offers the possibility to assess the safety-verification speed-up of checkpoint-driven fault-injection simulations. Until now, current technological limitations made this analysis impossible using other simulators with checkpointing features. This study uses VERITAS++ to generate SystemC models for the NanoMIPS core to measure this simulation speed-up. It performs three fault-injection campaigns on them. Each campaign

uses a fault library of 1000 transient faults (i.e., one bit-flip injection per simulation). Transient-fault injection is applied to randomly selected signals (i.e., over 14 K signals) and at random simulation times (i.e., up to 300 ns). In some cases, faults share the same injection time but are injected into different signals. Consequently, these faults also share the same checkpoint for fault injection. Each campaign uses a different distribution of fault-injection times. The first campaign concentrates them in the first half of the simulation. This scenario corresponds to systems which require long simulations to produce meaningful results. The second campaign focuses its fault-injection times on the second half of the simulation. This scenario corresponds to systems with long boot-up cycles, which also require simulation, before running the actual firmware code. The third campaign distributes its fault-injection times over the whole simulation. This scenario represents a generic HW system. Results show different speed-up values based on the distribution of injection times (Table 6.16). Faults injected closer to a simulation's end (e.g., in the second half of a simulation) benefit from the most significant speed-up. In this case, each fault-injection simulation starts at some point during the second half of the simulation and therefore avoids unnecessary simulation cycles (e.g., CPU boot-up, watchdog interrupts).

Table 6.16.: Safety-verification speed-up

A	B	C	D = B/C
Injection-Time Distribution Across the Simulation	Simulation Time Without Checkpointing	Simulation Time With Checkpointing	Speed-Up Factor
First half	1216.798	908.058	1.34x
Second half	1216.798	305.728	3.98x
Uniform	1216.798	593.560	2.05x
Harmonic Mean			2.02x

6.9. Summary

The average correlation factor of the analyzed VPs and gate-level net-lists is less than 13%. Therefore, VPs lack a significant number of fault-matching points to their equivalent gate-level net-lists. In turn, this limitation leads to a significant lack of fault-masking effects on such VPs and the observation of pseudo-fault effects. Furthermore, the three fault-injection tools introduced in this thesis have high performance when injecting single-bit and also multi-bit faults. For instance, SCFIT has, on average, 10-11% simulation overhead

when injecting a single fault, injectable TLM sockets have approximately 4-5% simulation overhead, and FIOs have the best performance with, on average, only 2% simulation overhead. SaVer uses spatial and temporal fault-pruning methods which detect and eliminate redundant fault injections. Furthermore, it enhances VPs with gate-level information using VERITAS and VERITAS++, two methods which have also been introduced in this thesis. Even though this approach sacrifices some of the original VP's simulation speed, the two methods improve the simulation speed compared to gate-level simulations, on average, by one order of magnitude. Moreover, these methods are up to six times faster than commercial simulators. Finally, SaVer's automated checkpointing approach achieved up to a 2x simulation speed-up, which has previously been technologically impossible. This speedup is now possible because SaVer saves checkpoints which are four orders of magnitude smaller than those from current state-of-the-art checkpointing methods. As a result of this reduction in size, SaVer also generates checkpoints up to one order of magnitude faster than existing checkpointing methods.

7. Conclusion

The goals of safety analysis and verification are to determine a system's fault tolerance, and develop appropriate safety-mitigation methods based on the system's safety requirements. However, as safety-critical SoCs steadily increase in complexity, more robust safety-verification methods are required to offer fast and accurate results.

In the attempt to provide fault-injection results early in a safety-critical SoC's development cycle, safety verification is being migrated from RTL models and gate-level net-lists to VPs. However, appropriate fault-injection methods are required at the VP level. This thesis has introduced three simulation-based, high-performance, and generic fault-injection methods for VPs developed using SystemC and TLM models. **SCFIT** has, on average, 10-11% simulation overhead when injecting a single fault using GDB. The **injectable TLM sockets** have 4-5% simulation overhead. The saboteur-based **FIOs** have the best performance, on average, with only 2% overhead.

Safety verification on VPs has two primary goals: the verification of safety mechanisms, and the calculation of a safety-critical SoC's fault-tolerance level. To achieve the first goal, safety-verification methods inject faults into SystemC and TLM-based VPs using direct fault injection. Then, simulation outcomes are analyzed to determine if the safety mechanism has detected and even corrected the injected faults. To achieve the second goal, faults are injected statistically (e.g., using Monte Carlo) into the SoC. Then, fault-injection outcomes are compared to a fault-free simulation to determine if an injected fault has caused system failures, has been masked by the system, detected, or corrected by a safety mechanism. These results are quantified to calculate the system's fault-tolerance level. However, as shown in this thesis, safety verification on VPs leads to inaccurate fault-injection results compared to fault injection into the SoC's gate-level net-lists. This inaccuracy is caused by the lack of implementation details on VPs compared to gate-level models, more explicitly missing fault-matching points and fault-masking effects. As already mentioned, VPs trade off these implementation details for faster simulation speed.

This thesis has quantified the number of fault-matching points, which are

fault-injection locations common across equivalent gate-level net-lists and VPs. Based on this quantification, even though the HW models are functionally equivalent, they only share fewer than 13% fault-matching points. Consequently, the correlation factor of VPs and gate-level net-lists is also low (i.e., less than 13%) because it decreases with the number of fault-matching points. Furthermore, a small correlation factor leads to high inaccuracy of fault-injection results, which leads to the observation of pseudo-fault effects on VPs (i.e., faults which cannot be reproduced on the gate-level net-list or physical implementation of the SoC). As a result, pseudo-fault effects can lead to the development of sub-optimal or even wrong safety mechanisms. Thus, better methods are needed to ensure quick and also accurate safety-verification results.

To address the shortcomings of missing fault-matching points, and to reduce the occurrence of pseudo-fault effects, this thesis has introduced two methods, called VERITAS and VERITAS++. These methods augment VPs with gate-level information by transforming combinational blocks (e.g., adders) and registers into behavioral models, which are then executed as part of the VPs. In doing so, these methods guarantee 100% correlation of VPs and gate-level net-lists. Furthermore, they improve the simulation speed of the original gate-level net-lists, on average, by a factor of two on adder architectures, and over one order of magnitude on a MIPS CPU core. Additionally, these methods produce VPs which simulate, on average, between 1.5x and over 5x faster than commercial HW simulators.

The simulation speed-up introduced by VERITAS and VERITAS++ has been further improved using multiple optimization methods. **Parallel simulation** allows the execution of multiple fault-injection simulations in parallel. Additionally, a **fault-collapsing** method has been developed, which reduces the number of redundant fault-injection locations, on average, by a factor of 2. A **fault-pruning** method has also been introduced in this thesis to decrease the system's fault-verification space even further. This method uses simulation-trace analysis to determine correct simulation times when injected faults can propagate through the analyzed system. On average, this fault-pruning method removes over 95% of faults from a fault library. Finally, a **checkpointing mechanism** speeds up safety-verification simulations approximately by a factor of 2. This speedup has been previously technologically impossible. This checkpointing mechanism requires, on average, four orders of magnitude less hard-disk space than the checkpoints from other safety-verification flows. Additionally, this method creates checkpoints up to six times faster than commercial simulators.

The three fault-injection methods, VERITAS, VERITAS++, and the optimiza-

tion methods presented above have been integrated into a safety-verification flow called SaVer. SaVer injects faults automatically into VPs developed using SystemC and TLM by using a fault library, which is created using data from safety-analysis methods (e.g., FMEDA). The outcomes of fault injection are determined by comparing fault-injection results with the results of a fault-free simulation (i.e., reference simulation). At the end of the fault-injection campaign, SaVer's simulation results can be used to calculate the analyzed system's fault tolerance. Furthermore, these results can be used to determine the accuracy of the failure rates predicted during safety analysis.

To sum up, the contributions presented in this thesis

- Enable early safety-architecture exploration on VPs.
- Overcome the challenges of fault-injection into VPs, such as missing fault-matching points and pseudo-faults.
- Speed-up fault-injection simulations.
- Bridge the gap between safety verification and safety analysis.

As a result, these contributions can be used to

- Obtain fast and accurate fault-injection results during the development process of safety-critical SoCs.
- Develop optimal safety mechanisms for safety-critical SoCs.
- Prove the accuracy of safety-analysis results predicted using FTA, DFA, and FMEDA.

Future work will focus on extending the tools and techniques presented in this thesis to analog circuits and embedded-software development. The long-term goal of this research is to create a safety-verification framework usable on any mixed-signal SoC, and thus, perform domain-independent safety verification. The short-term goal of this research is to implement additional automation approaches to SaVer, and further reduce the need for manual input from safety experts.

A. ISO 26262 and Functional Safety

A.1. Introduction

As cars become smarter, more interconnected, and enhanced with electronic and electric devices (e.g., GPS, ADAS, radar systems, video cameras), the complexity of guarantying a car's functionality (e.g., fault operational) also increases.

ISO 26262 defines functional safety as "the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electronic and electric systems" [5]. ISO 26262 applies to all electronic and electric and SW components built into vehicles up to 3.5 tons. It defines processes for manufacturing vehicles with the purpose of avoiding electronic and electric failures. The standard provides methods to prevent and control possible systematic and random HW failures, for instance, by defining safety requirements and development processes meant to reduce the risk of human injury or death.

ISO 26262 proposes an initial list of possible system faults in part 5, annex D, appendix 1 [5]. Fault models (e.g., offset, stuck in range) are introduced and classified based on the HW components they affect (e.g., relays, sensors, processing units). Furthermore, faults are attributed with safety-coverage metrics: low (60% of occurring HW faults), medium (90%), and high (99%), which can be achieved by enhancing the safety-critical system with safety mechanisms.

This section describes the notions of HW models, ASILs, timing requirements for safety-critical systems, and safety metrics introduced by ISO 26262.

A.2. Hardware Models

The safety standard uses HW models to assess a system's vulnerability towards faults (e.g., RTL models and gate-level net-lists). Models are defined as an abstraction of a physical system in which the system's inputs are mapped to its outputs. The model's abstraction level and granularity depend on the model's purpose. Thus, it is possible to create several models of the same system, which are used in different safety-critical analyses. Such models may be found on

various development layers of a safety-critical SoC, such as the application layer, the system layer, SW layer, the HW-architecture layer [160].

The completeness and granularity of a model represent a deciding factor to accurately assess a system's fault vulnerability [160]. However, specific models suffer from either under-specification or oversimplification. These models may lead to either too pessimistic or too optimistic results, which, in turn, may lead to the implementation of incorrect or unneeded safety mechanisms. Probabilistic models are examples of model incompleteness [160]. Here, experts may tend to use either failure catalogs (e.g., SN 29500) or incomplete field data to estimate a system's failure rate which may lead to overestimation or underestimation.

Models must be adequate for safety analysis regardless of their completeness and granularity levels [160]. In other words, the probability of residual failure within a model must have a realistic rate. However, this constraint can only be achieved if a model is specified with sufficient accuracy. In case of under-specification, the residual failure rate could still be low enough, but the system may continue to fail for other reasons unspecified in the model. Unfortunately, ISO 26262 provides no measures against such modeling insufficiencies (e.g., the insufficient granularity of a HW model).

A.3. Automotive Safety-Integrity Levels (ASILs)

ISO 26262 defines four safety compliance levels called ASILs, and it labels them with the letters A to D (the strictest). ASILs are valid over the system's entire lifecycle. ASILs define how safety-critical a system is (i.e., the system's maximum allowed FIT rate) and what type of safety measures have to be implemented in the system. Additionally, ASILs are given by three parameters (i.e., classes): (i) the severity (S) of a potential HW failure, (ii) the probability of exposure (E) to the failure, and (iii) the controllability (C) of an occurring failure. S determines a range of degrees of injury which the failure can inflict; the range spans from *no injury* (S0) to *life-threatening* (S3) (Table A.1). E describes the probability of exposure to an occurred failure and ranges from *incredible* (E0) to *high probability* (E4) (Table A.2). C characterizes how much control is lost in the event of failure occurrence. The interval varies from *controllable in general* (C0) to *difficult to control or uncontrollable* (C3) (Table A.3).

A system with a combination of sufficiently low S, E, C values (e.g., S2 E3 C2) is given ASIL A while a system with a combination of high S, E, C levels (i.e., S3 E4 C3) is assigned ASIL D. Table A.4 contains all possible ASIL combinations.

Table A.1.: Class of failure severity [5]

Severity	S0	S1	S2	S3
Description	No injuries	Light and moderate injuries	Severe and life-threatening injuries (survival probable)	Life-threatening injuries (survival uncertain), fatal injuries

Table A.2.: Class of exposure probability to a failure [5]

Exposure	E0	E1	E2	E3	E4
Description	Incredible	Very low probability	Low probability	Medium probability	High probability

Table A.3.: Class of controllability in case of failure [5]

Controllability	C0	C1	C2	C3
Description	Controllable in general	Simply controllable	Normally controllable	Difficult to control or uncontrollable

Furthermore, specific S-E-C configurations are attributed to quality management (QM) instead of an ASIL. QM requires compliance with quality-management standards such as ISO/TS 16949, ISO 9001, or equivalent.

A.4. Failures Modes

ISO 26262 introduces a failure-classification system based on failure causes and the system's ability to detect and correct them (Fig. A.1). ISO 26262 also introduces the term *safe fault*. A fault can be considered safe in two situations: either the analyzed HW element is not safety-relevant, or no fault or combination of two or more faults exists which can cause the system to fail. Since failure occurrence in non-safety-relevant HW elements does not lead to harm or loss of human life, it is not necessary to analyze fault effects on such elements. In the case of safety-relevant HW elements, a fault may propagate through the system on its own or by interacting with other faults. If a fault can propagate alone and if the system has no safety mechanism to detect or even correct it, the fault is called an single-point fault (SPF). Nevertheless, if a safety mechanism is already

Table A.4.: ASIL determination [5]

Severity Class	Probability Class	Controllability Class		
		C1	C2	C3
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

implemented but fails to prevent the failure, the fault becomes *residual*. However, upon detection, the fault becomes a *detected* multiple-point fault (MPF). If the HW does not detect the fault's effect, a human may still observe it. For instance, a driver may observe a problem with the car's steering wheel even if the car fails to signal this problem to the driver. Such a fault is called *perceived* MPF. Finally, if the fault's effect is neither detected nor perceived, it is called a *latent* MPF. MPFs can also occur if a fault only propagates through a system by interacting with other faults.

A.5. Tolerance-Time Interval

ISO 26262 defines a timing requirement for safety-critical systems called the tolerance-time interval (TTI). This interval describes the time allowed for the system to recover from a fault (Fig. A.2). The system recovers from faults by detecting them or their effects and then taking appropriate fault-mitigation actions (e.g., correct the fault, ignore the fault, reset the system, re-issue a command, enter a safe mode). Additionally, ISO 26262 defines the time allowed for fault detection as the diagnostic-test interval (DTI) and the time necessary to decide the severity of a fault as reaction time (RT). In general, the sum of TTI and RT must be smaller than TTI. Otherwise, the system is either unable to detect the fault or to recover from it. If the system detects the fault, reacts, and

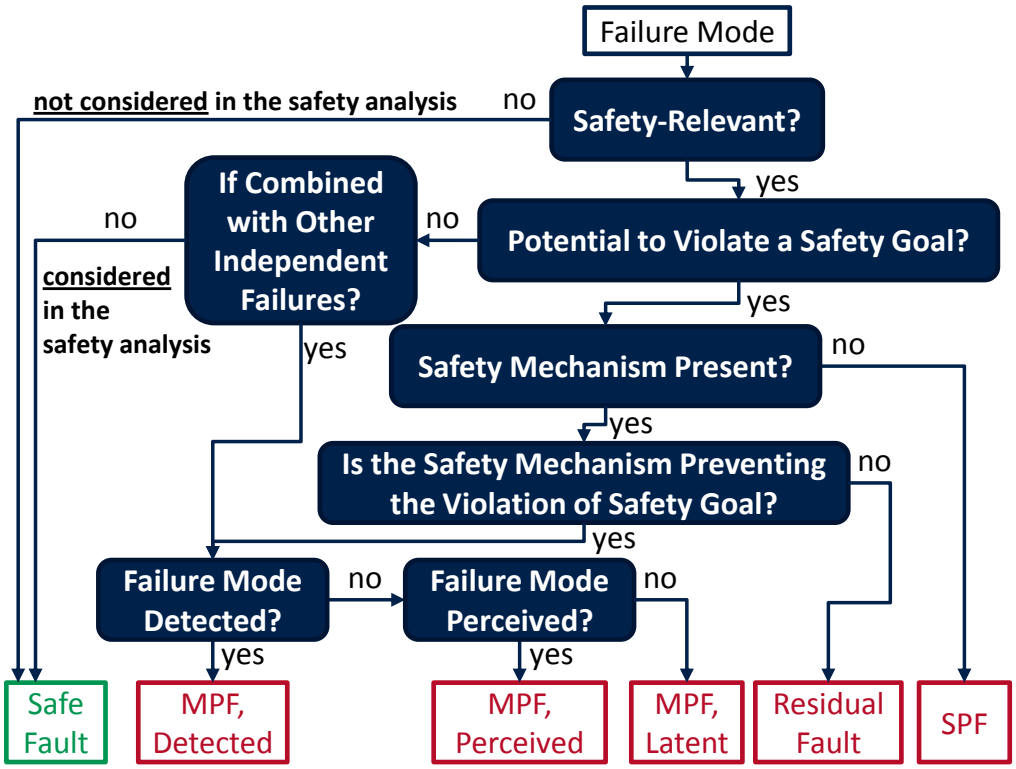


Figure A.1.: Failure-mode classification according to ISO 26262 [5]

enters in a safe state within the allotted TTI, the fault is considered as tolerated. Otherwise, ISO 26262 considers the fault as hazardous.

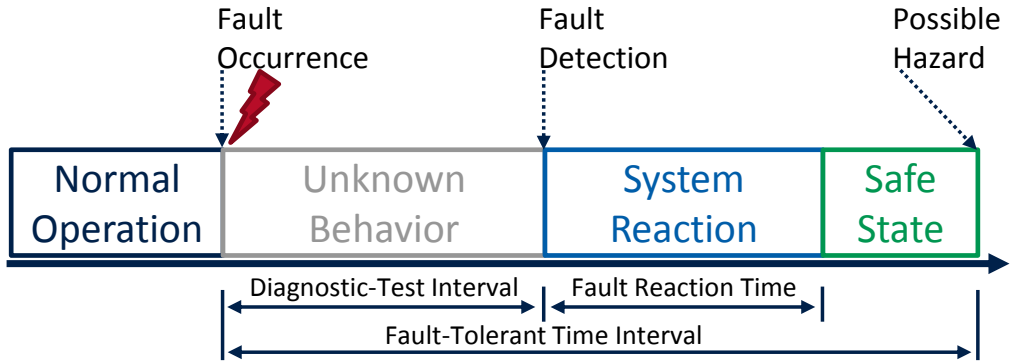


Figure A.2.: Diagnostic test interval, reaction time and tolerance time interval [5]

A.6. Safety-Coverage Metrics

Traditionally, safety coverage is determined using three metrics [83]:

Simulation coverage the number of simulations which lead to the observation of failures divided by the total number of simulations.

Error-detection coverage the number of simulations in which errors are detected divided by the total number of simulations.

Error-correction coverage the number of simulations in which errors are corrected divided by the total number of simulations.

ISO 26262 introduces two additional safety-coverage metrics: the single-point-fault metric (SPFM) and the latent-fault metric (LFM).

A.6.1. Single-Point-Fault Metric (SPFM)

The SPFM is a measure of how many different SPFs and unique residual faults (RFs) a safety-critical system can recover from. "A high single-point-fault metric implies that the proportion of single-point faults and residual faults in the hardware of the item is low" [5]. System robustness is provided either by the system's safety mechanisms or by its implicit design (e.g., fault tolerance of technology node, fault-masking effects, redundancy). The SPFM is given by the number of SPFs and RFs present in a safety-critical system divided by the system's total number of faults. Each ASIL defines a minimum number of SPFs from which a system must recover: ASIL B: 90%, ASIL C: 97%, and ASIL D: 99%.

A.6.2. Latent-Fault Metric (LFM)

The LFM "reflects the robustness of the item to latent faults either by coverage of faults in safety mechanisms or by the driver recognizing that the fault exists before the violation of the safety goal or by design (primarily safe faults). A high latent-fault metric implies that the proportion of latent faults in the hardware is low" [5]. The LFM is calculated as the number of perceived latent faults (LFs) and safe faults divided by the total number of faults present in the system. LFs are faults which are masked by the system for when running a specific application. However, this does not mean the faults are safe. For instance, MPFs can only lead to failures if additional faults are present. ISO 26262 only requires LF coverage of at least: 60% for ASIL B, 80% for ASIL C, and 90% for ASIL D.

B. Architecture Vulnerability Factor

The AVF is one of the most widely used analyses in the computer industry [17]–[22]. Since its introduction, it has been adapted into various branches, and has led to the creation of the AVF stack (Fig. B.1). This stack currently comprises the program vulnerability factor (PVF) [20], the operating-system vulnerability factor (OSVF) [23], the virtual-machine vulnerability factor (VMVF) [23], the HW vulnerability factor (HVF) [23], and the timing vulnerability factor (TVF) [24]. AVF is even used in the semiconductor industry to evaluate instruction-set architectures, where it is called the instruction vulnerability factor (IVF) [20].

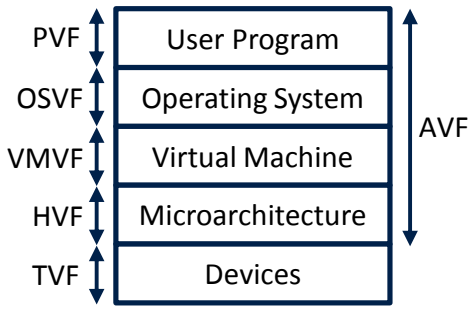


Figure B.1.: AVF stack across multiple branches in the computer industry [23]

In AVF analysis, the FIT rate of a system is given by the Cartesian product between the system's estimated AVF and its raw FIT rate obtained from technology catalogs, fault catalogs, or field samples. AVF is calculated as the sum of all architectural-correct-execution (ACE) bits within a system. An ACE bit is a system element (e.g., registers, flip-flops) which can cause a failure under the effect of a random HW faults. If the bit cannot produce a system failure, it is considered safe (i.e., un-ACE) [23].

In the computer industry, the resulting FIT rate estimates the system's fault vulnerability with sufficient accuracy [21]. However, the AVF calculation does not consider fault-masking effects. Furthermore, it assumes that all ACE bits have a uniform impact on the system. In reality, system elements switch between ACE and un-ACE states during execution. This switch is mainly influenced by the system's configuration and its workloads. Consequently, some bits exhibit

more ACE states than others. A status register, for example, may be polled (i.e., read) more times than a configuration register which is accessed only once during system start-up. As a result, the status register has more ACE states than the configuration register, and thus, a higher probability of causing a system failure. Therefore, this method of AVF calculation is not sufficiently accurate to verify safety-critical systems.

The AVF accuracy of safety-critical systems can be improved by establishing the criticality of each ACE bit. This criticality can be determined using life-time analysis, which measures the fraction of simulation time in which system elements are in ACE states (i.e., when injected faults can cause failures) and un-ACE states (i.e., when injected faults are masked by the system). Thus, AVF can be calculated as the ratio between the sum of simulation-time intervals of all ACE bits of a system and the total simulation time [23]:

$$AVF = \frac{\sum_{i=0}^{n} t_{sim}^{ACE-bit_i}}{t_{sim}} \tag{B.1}$$

Despite this improvement, AVF suffers from a series of drawbacks. **First**, technologies smaller than 90 nm require safety verification of temporal and spatial multi-bit faults. However, when determining FIT rates for multi-bit faults, AVF overestimates these results sevenfold [22]. **Second**, the abstract behavioral models used to compute AVF do not contain all the necessary information available within detailed RTL or gate-level models. For this reason, AVF is 2-3x less reliable than SFI [25]. Furthermore, the reduction of AVF for some system components may even increase the AVF of others without obtaining an overall benefit [26]. **Third**, the current definition of ACE bits does not take into account all safety requirements. For instance, the occurrence of a potentially catastrophic fault in a car's ECU may lead the ECU to interpret the *brake* command as an *accelerate* command. In the computer industry, this is not a fault scenario since both the *break* and *accelerate* commands are valid within the ECU. However, this is a critical fault scenario in the safety domain. Such a fault would cause the car to speed up instead of slowing down, which is a deviation from the driver's intention, and thus, from the system's specified behavior. Furthermore, if the driver is braking to stop at a busy intersection or to exit the freeway, this fault could potentially endanger the passengers' lives. Consequently, the definition of ACE bits must be redefined to also focus on such safety requirements and the AVF estimation accuracy must be improved to avoid such safety-critical misbehaviors.

C. Linking Safety Analysis to Fault-Injection Frameworks

To avoid "unnecessary risk" [5], ISO 26262 requests the prediction of potential risks using high-level safety analyses, which can be performed on safety-critical systems using FTAs [117], DFAs [117], and FMEDAs [119]. **FTA** is a qualitative (deductive) analysis performed top-down starting from problematic situations on the system level and looking for the similar causes within the system's components [122]. **DFA** is a qualitative analysis which focuses on failures whose occurrence is interdependent (also called *common-cause failures* or *cascading failures*); these failure modes either have the same cause or their occurrence necessarily triggers another subsequent failure. **FMEDA** is a quantitative (inductive) approach which starts bottom-up from local component malfunctions moving towards the corresponding effects on the complete system [118].

These system-level analyses help to predict a system's FIT rate and drive the development of safety mechanisms, such as error-correction code (ECC), lockstep, or redundant registers. However, as the complexity of safety-critical systems increases, the manual effort of performing such analyses also increases [121]. Furthermore, FTAs, DFAs, and FMEDAs are not evaluated on an actual safety-critical system. Instead, they only *predict* a system's FIT rate. The system's *real* FIT rate is then assessed on the manufactured product or by performing fault-injection-based safety verification. However, even in this case, high-level safety analyses and fault-injection experiments are performed independently from each other. Consequently, this decoupling hinders safety verification refinement and also limits the re-usability of FTAs, DFAs, and FMEDAs on future products [117]. These drawbacks can be mitigated by linking safety analyses with fault-injection methods [119], [121]. In doing so, FIT rates predicted using high-level safety analysis become significantly refined with statistical results (i.e., estimated FIT rates) obtained from fault-injection simulations.

Before linking system-level safety analyses to fault-injection frameworks, the structure of FMEDAs, DFAs, and FTAs must first be formalized (e.g., databases, spreadsheets). This step allows software tools to parse and analyze the data

contained therein (e.g., system information, failure modes, safety mechanisms) (Fig. C.1). For instance, the authors of [121] used model-based frameworks to formalize the contents of FMEDAs, DFAs, and FTAs. Here, additional parsers are needed to extract information from a system's implementation (e.g., VPs, RTL models, gate-level net-lists). The extracted data (e.g., modules, sub-modules, flip-flops, logic gates) are mapped to relevant safety functions from the FMEDA, DFA, and FTA to define the HW system's fault-verification space. In other words, the mapping step between the safety-analysis and the fault-simulation platforms sets up fault-injection points, observation points, fault types (e.g., stuck-at faults, bit-flips) and other attributes necessary for executing fault-injection simulations. Next, the data collected within the data models are used to generate fault libraries which are then utilized by safety-verification flows (e.g., SaVer). Then, the simulation results provided by fault-injection simulations (e.g., diagnostic coverage, passed/failed tests) are back-annotated to the FMEDAs and FTAs data models. Finally, FIT rates, which were only predicted at first, become refined by the fault-injection simulation results.

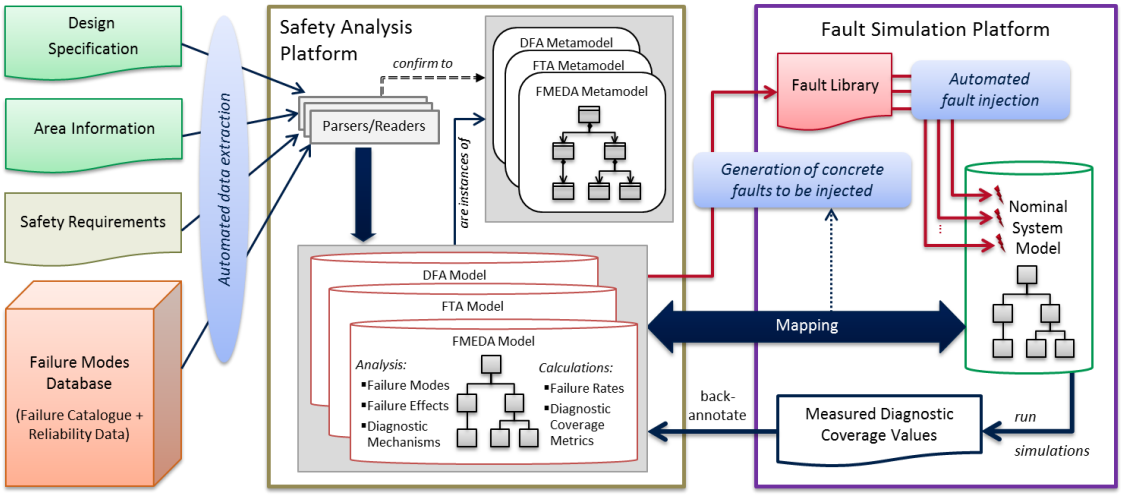


Figure C.1.: Model-based safety analysis and link to fault simulation [119]

The link between safety analysis and the HW system takes advantage of the correspondence among *basic safety-evaluation features* (e.g., targets, threats, and counter-measures) and *basic HW-system features* (e.g., modules, sub-modules, entities, components) (Fig. C.2). Consequently, safety targets such as parts and functions in the FMEDA as well as elements in the DFA are mapped to modules, sub-modules, entities, components, and other elements from the safety-critical system. Threats are classified differently based on the employed safety analysis

and their mapping on the system level. First, they are classified as failure modes in FMEDAs and dependent failures in DFAs and are mapped to fault-injection points such as signals, ports, sockets, etc. Second, they are classified as failure effects in FMEDAs and DFAs which are mapped to observation points. Third, they are classified as events in FTAs, which are mapped either to injection or observation points based on their functionality. Counter-measures are defined as safety measures in FMEDAs and DFAs and are mapped to diagnostic and correction points. The implementation of these measures can be done either with additional modules or by using members of already existing modules (e.g., HW signals).

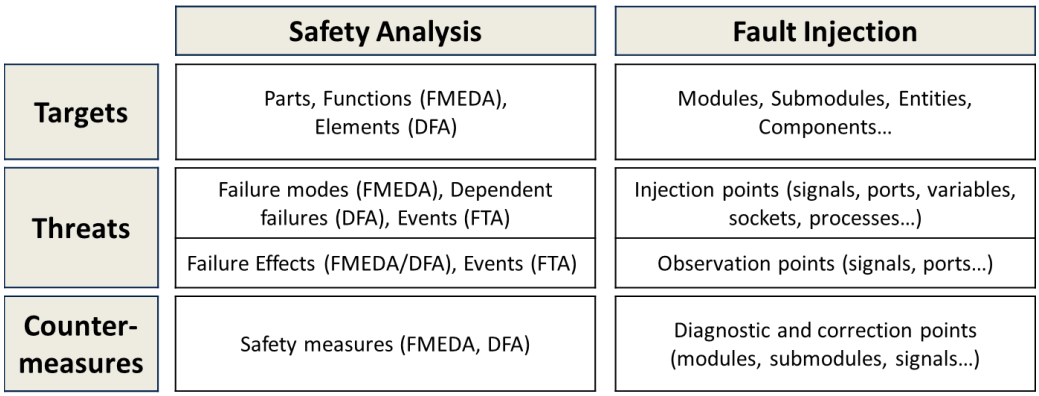


Figure C.2.: Data mapping from safety analysis to the HW system [119]

Mapping safety-analysis information to the HW system takes place through expression and pattern matching (Fig. C.3). This step is necessary because the safety analysis and system-level modeling take place in parallel based on the available (safety) requirements and development specification. The mapping step can be performed in two ways and begins once both safety analysis and

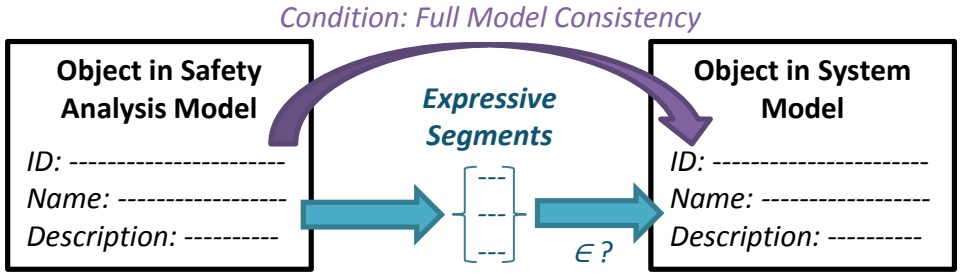


Figure C.3.: Conditions for data mapping from safety analysis to fault injection [119]

the HW are stable. First, ID-based matching is useful if safety-analysis data are kept consistent with data from system modeling. In this case, specific IDs must be defined in the development specification for modules, sub-modules, signals, parts, functions, etc. These IDs are then used consistently throughout the system and safety analysis. Second, name-based matching is used instead of IDs. In this case, particular names, descriptions, and other attributes from the system (e.g., signal paths) and safety analysis (e.g., failure-mode names) are preprocessed and filtered to identify so-called *expressive segments*. Thus, the user is provided with a flexible tool to map the correct safety information to the safety-critical system.

Acronyms

ACE Architectural Correct Execution

ADAS Advanced Driver-Assistance Systems

ALU Arithmetic-Logic Unit

API Application Programming Interface

ASIL Automotive Safety-Integrity Level

ATPG Automatic Test-Pattern Generation

AVF Architecture-Vulnerability Factor

CAS Cycle-Accurate Simulator

CMOS Complementary Metal-Oxide-Semiconductor

COI Cone of Influence

CPU Central Processing Unit

CSV Comma-Separated Value

DFA Dependent-Failure Analysis

ECC Error-Correction Code

ECU Engine-Control Unit

EDS Event-Driven Simulator

FIO Fault-Injection Object

FIT Failure in Time

FMEDA Failure Modes, Effects, and Diagnostics Analysis

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

FTA Fault-Tree Analysis

GDB GNU Debugger

GPU Graphics-Processing Unit

GUI Graphical User Interface

HDL Hardware-Description Language

HW Hardware

LF Latent Fault

LFM Latent-Fault Metric

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor

MPF Multiple-Point Fault

PC Personal Computer

QM Quality Management

RF Residual Fault

RT Reaction Time

RTL Register-Transfer Level

SFI Statistical Fault Injection

SoC System on Chip

SPF Single-Point Fault

SPFM Single-Point-Fault Metric

SW Software

TLM Transaction-Level Modeling

TTI Diagnostic-Test Interval

TTI Tolerance-Time Interval

UDFI User-Defined Fault Injection

VAD Value-Access Dump

VCD Value-Change Dump

VM Virtual Machine

VP Virtual Prototype

Glossary

Fault-Matching Point Fault-injection locations present across two or more abstraction levels (e.g., VPs and gate-level models). 45, 49–53, 59, 80–82, 98, 101–103

ISO 26262 The automotive safety standard applied on all electronic-and-electric components built into vehicles up to 3.5 t. vii, 1–3, 5, 6, 16, 22, 47, 49, 105–110, 115, 125, 126

Latent Fault Multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver within a predetermined time interval. 120

Latent-Fault Metric Reflects the robustness of the item to latent faults either by coverage of faults in safety mechanisms or by the driver recognizing that the fault exists before the violation of the safety goal, or by design (primarily safe faults). A high latent-fault metric implies that the proportion of latent faults in the hardware is low. 120

Multiple-Point Fault A SPF which leads to a failure when combined with other independent SPFs. 120

Pseudo-Failure The effects of pseudo-fault injection observed at the outputs of virtual prototypes. vii, viii, 45, 47, 49, 51, 52, 59

Pseudo-Fault Single-bit or multiple-bit faults injected on the VP abstraction level which lead to system failures but whose effects cannot be reproduced on the gate abstraction level through injection of single-bit faults. 45, 47, 49, 50, 52, 53, 94, 98, 102, 103, 123

Residual Fault Portion of a hardware or software fault which by itself leads to the violation of a safety requirement. 120

SaVer Safety-verification flow optimized for the analysis of effects of random HW faults on VPs. viii, 17, 18, 61, 64–68, 71, 73, 75, 90–93, 97, 99, 103, 116, 126, 129

SCFIT A fault-injection tool for SystemC and TLM models developed under any abstraction level. 33–40, 44, 56, 58, 87–89, 98, 101, 125, 129

Single-Point Fault A hardware or software fault which is not covered by a safety mechanism and that leads directly to the violation of a safety requirement. 120

Single-Point-Fault Metric Reflects the robustness of the item to single-point and residual faults either by coverage from safety mechanisms or by design (primarily safe faults). A high single-point fault metric implies that the proportion of single-point faults and residual faults in the hardware of the item is low. 120

System on Chip A system which is fully integrated onto a single chip (e.g., single or multiple cores, high-speed bus, a peripheral bus, several dedicated HW peripherals). 120

SystemC A class library on top of C++ which supports HDL concepts for modeling concurrency and communication. vii, 7–10, 15–21, 26, 33–41, 43, 44, 53–58, 64–68, 77, 78, 80, 87, 89, 92, 97, 101, 103, 123

VERITAS A tool which transforms Verilog net-lists to C++, SystemC, TLM code. 17, 18, 53–57, 59, 78, 81, 82, 89, 93–97, 99, 102, 126, 129

VERITAS++ An extension brought to VERITAS. 17, 18, 53, 56–59, 65, 67, 78, 81, 82, 89, 92, 93, 96, 97, 99, 102, 126, 129

List of Figures

1.1. ISO 26262's simplified V-model diagram 3
1.2. Generic checkpoint-restore mechanism 13

2.1. Simple SystemC model 20
2.2. Basic TLM model 21
2.3. TLM-SystemC wrapper module which synchronizes the communication across SystemC and TLM models 21
2.4. Generic test-bench framework 22
2.5. Generic fault-injection testbench [41] 23
2.6. Fault verification space [89] 24
2.7. Horizontal fault-error-failure propagation chain [83] 26
2.8. Vertical fault-error-failure propagation chain 26

3.1. SCFIT's fault-injection mechanism 32
3.2. SCFIT's fault-injection execution flow 35
3.3. SCFIT's fault-injection flow 35
3.4. Example of SCFIT's GUI 36
3.5. Example of SCFIT's generated fault-injection report 37
3.6. Registration of fault-injection objects to fault-injection-object manager and access from testbench 38
3.7. TLM injectable interface 39
3.8. TLM injectable sockets 41

4.1. Schematics of special_logic Verilog module 47
4.2. Graph-based representation of missing fault-matching points across gate-level net-lists and VPs 49
4.3. Graph-based representation of fault-propagation paths on gate-level net-lists and VPs 50
4.4. VERITAS flow diagram 52
4.5. VERITAS C++ representation of a full adder 52
4.6. VERITAS breakdown of multi-bit operations 53

4.7. Extension of VERITAS' C++ generator to support fault-injection objects 54
4.8. VERITAS++ flow diagram 55
4.9. Verilator flow diagram 55

5.1. Saturation curve for the number of fault-injection simulations n given by the number of faults N ($p = 0.5$, $e = 1\%$, and $t = 3.0902$ for 99.8% confidence interval) 61
5.2. SaVer's fault-injection regression flow 62
5.3. VAD format compared to VCD format 64
5.4. Difference between masked faults and faults which propagate based on read-write accesses on registers 65
5.5. VCD-based checkpointing-generation flow 67
5.6. UML diagram of meta-model structure 68
5.7. VCD-based restoring-from-checkpoint flow 69
5.8. SaVer's checkpointing-based fault-injection regression flow . . . 71

6.1. Block diagram of NanoMIPS' SystemC, RTL, and gate-level model 77
6.2. Block diagram of NanoMIPS' loosely-time TLM model 77
6.3. Block diagram of NanoMIPS' approximately-time TLM model . 78
6.4. Observed number of faulted bit positions in the sum output of an 8, 16, and 32-bit adder after 10 000 samples 81
6.5. Coverage of carry-out failures per fault type observed for 8, 16, and 32-bit adder 82
6.6. Maximum multi-bit failure observed after injecting stuck-at-1 faults in adders of variable sizes 82
6.7. Maximum multi-bit failure observed after injecting stuck-at-0 faults in adders of variable sizes 83
6.8. Single-bit failures observed for various MIPS CPU gate-level sub-blocks after injecting permanent faults into 10 000 random samples 85
6.9. Multi-bit failures observed for several MIPS CPU gate-level sub-blocks after injecting stuck-at-1 faults 85
6.10. Simulation-speed comparison between each gate-level and gate-level-accurate TLM 93

A.1. Failure-mode classification according to ISO 26262 [5] 107
A.2. Diagnostic test interval, reaction time and tolerance time interval [5] 107

B.1. AVF stack across multiple branches in the computer industry [23] 109

C.1. Model-based safety analysis and link to fault simulation [119] . . 112
C.2. Data mapping from safety analysis to the HW system [119] . . . 113
C.3. Conditions for data mapping from safety analysis to fault injection [119] 113

List of Tables

2.1. Example of fault-model classifications based on abstraction level 28

3.1. SCFIT's requirements for placing breakpoints and watchpoints . 32

4.1. Single-bit failures observed on special_logic module 46
4.2. Double-bit failures observed on special_logic module 46
4.3. Failures caused by 2-bit faults injected into special_logic_schematic2 48

6.1. Adders–Correlation factor across the VP and gate abstraction levels 79
6.2. NanoMIPS–Correlation factor across the VP and gate abstraction levels 80
6.3. Average number of simulations required to cause a maximum multi-bit failure in several adders 84
6.4. Simulation overhead measured for SCFIT 86
6.5. Simulation overhead measured for injectable TLM sockets 87
6.6. Hard-disk-space requirements for one checkpoint 88
6.7. Hard-disk-space requirements for 1000 checkpoints 89
6.8. Generation time for 1000 checkpoints 89
6.9. VERITAS++–Spatial fault pruning 90
6.10. SaVer–Temporal fault pruning 91
6.11. Adders–Simulation overhead introduced by VERITAS 92
6.12. Adders–Simulation performance vs improved fault-injection correlation 93
6.13. NanoMIPS–Simulation performance introduced by VERITAS . . 94
6.14. NanoMIPS–Simulation speed vs improved fault-injection correlation 95
6.15. Simulation performance of VERITAS++ 95
6.16. Safety-verification speed-up 96

A.1. Class of failure severity [5] 105
A.2. Class of exposure probability to a failure [5] 105
A.3. Class of controllability in case of failure [5] 105
A.4. ASIL determination [5] 106

Bibliography

[1] S. Choi, F. Thalmayr, D. Wee, and F. Weig, *Advanced Driver-Assistance Systems: Challenges and Opportunities Ahead*, Accessed on 02.01.2018, 11:15, Feb. 2016. [Online]. Available: https://www.mckinsey.com/industries/semiconductors/our-insights/advanced-driver-assistance-systems-challenges-and-opportunities-ahead.

[2] K. Korosec, *Intel Predicts a $7 Trillion Self-Driving Future*, Accessed on 02.01.2018, 11:22, Jun. 2017. [Online]. Available: https://www.theverge.com/2017/6/1/15725516/intel-7-trillion-dollar-self-driving-autonomous-cars.

[3] N. Bunkley, *Toyota Settles Over California Deaths*, Accessed: 2015-11-10, 17:53, Sep. 2010. [Online]. Available: https://www.nytimes.com/2010/09/19/business/19autos.html.

[4] P. Koopman, *A Case Study of Toyota Unintended Acceleration and Software Safety*, Accessed: 2018-06-20, 17:24, Sep. 2014. [Online]. Available: https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf.

[5] ISO, CD, "26262, Road Vehicles–Functional Safety," *International Standard ISO/FDIS*, vol. 26262, 2011.

[6] R. R. Schaller, "Moore's Law: Past, Present and Future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[7] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs from a 250 nm to a 22 nm Design Rule," *Electron Devices, IEEE Transactions on*, vol. 57, no. 7, pp. 1527–1538, 2010.

[8] G. Rangarajan and J. Deng, *Addressing Signal Electromigration (EM) in Today's Complex Digital Designs*, Accessed on 12.04.2017, 15:00, Jan. 2013. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1280370.

[9] A. Herkersdorf, M. Engel, M. Glaß, J. Henkel, V. B. Kleeberger, M. Kochte, J. M. Kühn, S. R. Nassif, H. Rauchfuss, W. Rosenstiel, *et al.*, "Cross-Layer Dependability Modeling and Abstraction in System on Chip," in *Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2013.

[10] R. C. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[11] V. V. Kumar and J. Lach, "Fine-Grained Self-Healing Hardware for Large-Scale Autonomic Systems," in *Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on*, IEEE, 2003, pp. 707–712.

[12] E. Ibe, K.-i. Shimbo, H. Taniguchi, T. Toba, K. Nishii, and Y. Taniguchi, "Quantification and Mitigation Strategies of Neutron Induced Soft-Errors in CMOS Devices and Components," in *Reliability Physics Symposium (IRPS), 2011 IEEE International*, IEEE, 2011, pp. 3C–2.

[13] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn, "Reliable On-Chip Systems in the Nano-Era: Lessons Learnt and Future Trends," in *Proceedings of the 50th Annual Design Automation Conference*, ACM, 2013, p. 99.

[14] J. C. Grebe and W. M. Goble, "FMEDA–Accurate Product Failure Metrics," *FMEDA Development Paper, Rev*, vol. 1, 2007.

[15] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault Tree Handbook," Nuclear Regulatory Commission Washington dc, Tech. Rep., 1981.

[16] L. Stein, *ISO 26262 Part 11 - Blog Post 3: Dependent Failure Analysis (DFA)*, Accessed on 05.01.2017, 11:34, Jul. 2017. [Online]. Available: `https://lorit-consultancy.com/en/2017/07/iso-26262-part-11-blog-post-3-dependent-failure-analysis-dfa/`.

[17] M. Maniatakos, C. Tirumurti, A. Jas, and Y. Makris, "AVF Analysis Acceleration via Hierarchical Fault Pruning," in *European Test Symposium (ETS), 2011 16th IEEE*, IEEE, 2011, pp. 87–92.

[18] M. Maniatakos, M. K. Michael, and Y. Makris, "Investigating the Limits of AVF Analysis in the Presence of Multiple Bit Errors," in *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, IEEE, 2013, pp. 49–54.

[19] S. Mittal and J. Vetter, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems," *IEEE Transaction on Parallel and Distributing Systems*, vol. 27, no. 4, pp. 1226–1238, 2016.

[20] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011.

[21] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2003, p. 29.

[22] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient Fault Models and AVF Estimation Revisited," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, IEEE, 2010, pp. 477–486.

[23] V. Sridharan and D. R. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 38, 2010, pp. 461–472.

[24] N. Seifert and N. Tam, "Timing Vulnerability Factors of Sequentials," *IEEE Transactions on Device and Materials Reliability*, vol. 4, no. 3, pp. 516–522, 2004.

[25] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE Analysis Reliability Estimates using Fault Injection," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 35, 2007, pp. 460–469.

[26] L. Duan, Y. Zhang, B. Li, and L. Peng, "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, IEEE, 2011, pp. 247–256.

[27] R. A. Howard, *Dynamic Probabilistic Systems: Markov Models*. Courier Corporation, 2012, vol. 1.

[28] E. Karimi, M. H. Haghbayan, A. Maleki, and M. Tabandeh, "Graph Based Fault Model Definition for Bus Testing," in *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013.

[29] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing," *IEEE Transactions on Nuclear Science*, vol. 60, no. 3, pp. 2119–2142, 2013.

[30] P. Maistri and R. Leveugle, "Towards Automated Fault Pruning with Petri Nets," in *On-Line Testing Symposium, 2009. IOLTS 2009. 15th IEEE International*, IEEE, 2009, pp. 41–46.

[31] F. Kastensmidt and P. Rech, *FPGAs and Parallel Architectures for Aerospace Applications: Soft Errors and Fault-Tolerant Design*. Springer, 2015.

[32] A. G. Schmidt, B. Huang, R. Sass, and M. French, "Checkpoint/Restart and Beyond: Resilient High Performance Computing with FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, IEEE, 2011, pp. 162–169.

[33] M. Alderighi, F. Casini, S. d'Angelo, M. Mancini, S. Pastore, and G. R. Sechi, "Evaluation of Single Event Upset Mitigation Schemes for SRAM Based FPGAs Using the FLIPPER Fault Injection Platform," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*, IEEE, 2007, pp. 105–113.

[34] F. Lima, C Carmichael, J Fabula, R Padovani, and R. Reis, "A Fault Injection Analysis of Virtex FPGA TMR Design Methodology," in *Radiation and Its Effects on Components and Systems, 2001. 6th European Conference on*, IEEE, 2001, pp. 275–282.

[35] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, IEEE, 2014, pp. 221–230.

[36] T. S. Tan and B. A. Rosdi, "Verilog HDL Simulator Technology: A Survey," *Journal of Electronic Testing*, vol. 30, no. 3, pp. 255–269, 2014.

[37] R Rajaraman, J. Kim, N. Vijaykrishnan, Y. Xie, and M. J. Irwin, "SEAT-LA: A Soft Error Analysis Tool for Combinational Logic," in *19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06)*, IEEE, 2006, 4–pp.

[38] J.-S. Chang and C.-S. Lin, "Test set compaction for combinational circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 14, no. 11, pp. 1370–1378, 1995.

[39] I. Hamzaoglu and J. H. Patel, "Test set compaction algorithms for combinational circuits," in *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, ACM, 1998, pp. 283–289.

[40] V. Sieh, O. Tschäche, and F. Balbach, "VHDL-Based Fault Injection with VERIFY," *Interner Bericht*, vol. 5, p. 96, 1996.

[41] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[42] I. Stefanovici, A. Hwang, and B. Schroeder, *DRAM's Damning Defects–and How They Cripple Computers*, Accessed on 06.06.2016, 15:43, Nov. 2015. [Online]. Available: `http://spectrum.ieee.org/computing/hardware/drams-damning-defects-and-how-they-cripple-computers`.

[43] M. Raji, B. Ghavami, and H. Pedram, "Gate Resizing for Soft Error Rate Reduction in Nano-scale Digital Circuits Considering Process Variations," in *Digital System Design (DSD), 2015 Euromicro Conference on*, IEEE, 2015, pp. 445–452.

[44] N. George and J. Lach, "Characterization of Logical Masking and Error Propagation in Combinational Circuits and Effects on System Vulnerability," in *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, IEEE, 2011, pp. 323–334.

[45] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[46] I.-D. Vidrascu, "Implementation of a Safety Verification Environment (SVE) Based on Fault Injection," Master's thesis, Fachhochschule Kärnten, Klagenfurt am Wörthersee, Austria, 2015.

[47] A. K. Ghosh, T. A. DeLong, B. W. Johnson, and J. A. Profeta III, "Fault Injection in the Design Process Using VHDL," in *VHDL International Users' Forum Fall Conference*, 1995, pp. 15–19.

[48] H. Ziade, R. A. Ayoubi, R. Velazco, *et al.*, "A Survey on Fault Injection Techniques," *Int. Arab J. Inf. Technol.*, vol. 1, no. 2, pp. 171–186, 2004.

[49] S. Tixeuil, W. Hoarau, and L. Silva, "An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids," in *Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, 2006.

[50] R. Mariani, G. Boschi, and F. Colucci, "Using an Innovative SoC-Level FMEA Methodology to Design in Compliance with IEC61508," in *Proceedings of the conference on Design, automation and test in Europe*, EDA Consortium, 2007, pp. 492–497.

[51] K. Greb, A. Arora, and R. Yogitech, *Tool For Automation Of Functional Safety Metric Calculation And Prototyping Of Functional Safety Systems*, US Patent App. 14/020,802, Jun. 2014. [Online]. Available: `https://www.google.com/patents/US20140173548`.

[52] A. Benso, A. Bosio, S. Di Carlo, and R. Mariani, "A Functional Verification Based Fault Injection Environment," in *Defect and Fault-Tolerance in VLSI Systems, 2007. DFT'07. 22nd IEEE International Symposium on*, IEEE, 2007, pp. 114–122.

[53] Cadence, *Functional Safety*, Accessed: 12.01.2017, 2016. [Online]. Available: `https://www.cadence.com/content/cadence-www/global/en_US/home/solutions/automotive-solution/functional-safety.html`.

[54] Synopsys, *Automotive Safety Verification for ISO 26262*, Accessed: 12.01.2017, 2016. [Online]. Available: `https://www.synopsys.com/verification/automotive-safety-verification-for-iso-26262.html`.

[55] Infineon Technologies AG, *AURIX–TriCore Datasheet*, Accessed: 2016-2-22. [Online]. Available: `https://www.infineon.com`.

[56] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann, A. Uhlig, U. Grätz, and R. Lien, "Engineering Failure Analysis and Design Optimisation with HiP-HOPS," *Engineering Failure Analysis*, vol. 18, no. 2, pp. 590–608, 2011.

[57] Y. Xiong, B. Qin, M. Wu, J. Yang, and M. Fan, "LabVIEW AND MATLAB-Based Virtual Control System for Virtual Prototyping of Cyclotron," in *Particle Accelerator Conference, 2007. PAC. IEEE*, IEEE, 2007, pp. 281–283.

[58] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull, "Interactive Presentation: Implementation of a Transaction Level Assertion Framework in SystemC," in *Proceedings of the conference on Design, automation and test in Europe*, EDA Consortium, 2007, pp. 894–899.

[59] S. A. Misera, "Simulation von Fehlern in Digitalen Schaltungen mit SystemC," PhD thesis, Universitätsbibliothek, 2007.

[60] S. Misera, H. T. Vierhaus, and A. Sieber, "Fault Injection Techniques and Their Accelerated Simulation in SystemC," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, IEEE, 2007, pp. 587–595.

[61] A. Fin, F. Fummi, and G. Pravadelli, "SystemC as a Complete Design and Validation Environment," in *SystemC*, Springer, 2003, pp. 127–156.

[62] A. Fin and F. Fummi, "Laerte++: An Object Oriented High-Level TPG for SystemC Designs," in *Languages for system specification*, Springer, 2004, pp. 105–117.

[63] W. Lu and M. Radetzki, "Efficient Fault Simulation of SystemC Designs," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, IEEE, 2011, pp. 487–494.

[64] F. Rogin, E. Fehlauer, C. Haufe, and S. Ohnewald, "Debug Patterns for Efficient High-level SystemC Debugging," in *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS'07. IEEE*, IEEE, 2007, pp. 1–6.

[65] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "SystemC-Based Minimum Intrusive Fault Injection Technique with Improved Fault Representation," in *On-Line Testing Symposium, 2008. IOLTS'08. 14th IEEE International*, IEEE, 2008, pp. 99–104.

[66] F. Bruschi, F. Ferrandi, and D. Sciuto, "A Framework for the Functional Verification of SystemC Models," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 667–695, 2005.

[67] J. Vennin, S. Penain, L. Charest, S. Meftali, and J.-L. Dekeyser, "Embed scripting inside systemc," in *FDL*, 2005, pp. 373–385.

[68] F. Arlati-arlati, A. Miele, and F. Bruschi, "ReSP User Manual," Revision 2: June 2011.

[69] C. Bolchini, A. Miele, and D. Sciuto, "Fault Models and Injection Strategies in SystemC Specifications," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*, IEEE, 2008, pp. 88–95.

[70] G. Beltrame and L. Fossati, "ReSP: A Design and Validation Tool for Data Systems," in *DASIA 2008 Data Systems In Aerospace*, vol. 665, 2008, p. 22.

[71] G. Beltrame, L. Fossati, and D. Sciuto, "ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1857–1869, 2009.

[72] V. Guarnieri, N. Bombieri, G. Pravadelli, F. Fummi, H. Hantson, *et al.*, "Mutation Analysis for SystemC Designs at TLM," in *Test Workshop (LATW), 2011 12th Latin American*, Mar. 2011, pp. 1–6.

[73] K.-J. Chang and Y.-Y. Chen, "System-Level Fault Injection in SystemC Design Platform," in *Proceedings of 8th International Symposium on Advanced Intelligent Systems (ISIS)*, Citeseer, 2007.

[74] STMicroelectronics, *32-bit Power Architecture® Microcontroller for Automotive SIL3/ASIL-D Chassis and Safety Applications*, SPC56 Datasheet, Rev 11, 2014.

[75] A. d. Silva, P. Parra, Ó. R. Polo, and S. Sánchez, "Runtime Instrumentation of SystemC/TLM2 Interfaces for Fault Tolerance Requirements Verification in Software Cosimulation," *Modelling and Simulation in Engineering*, vol. 2014, p. 42, 2014.

[76] Cadence, *Preliminary e Language Reference Draft*, Accessed: 29.12.2017, 2003. [Online]. Available: `http://rise.cse.iitm.ac.in/people/faculty/kama/prof/eLRM1.pdf`.

[77] L. Duan and W.-M. Lin, "Hardware vs. Simulation: Bridging the Gap between Efficiency and Flexibility for Computer Engineering Education and Research," in *ASEE Gulf-Southwest Annual Conference*, 2015, pp. 1–6.

[78] D. Mueller-Gritschneder, M. Greim, and U. Schlichtmann, "Safety Evaluation Based on Virtual Prototypes: Fault Injection with Multi-Level Processor Models," in *Integrated Circuits (ISIC), 2016 International Symposium on*, IEEE, 2016, pp. 1–2.

[79] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," in *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, IEEE, 2013, pp. 1–10.

[80] J. Espinosa, D. de Andrés, J. C. Ruiz, C. Hernandez, and J. Abella, "Towards Certification-Aware Fault Injection Methodologies Using Virtual Prototypes," *Forum on Specification and Design Languages (FDL) – Work in Progress (WiP)*, pp. 1–4, 2015.

[81] J. Espinosa, C. Hernandez, and J. Abella, "Characterizing Fault Propagation in Safety-Critical Processor Designs," in *On-Line Testing Symposium (IOLTS), 2015 IEEE 21st International*, IEEE, 2015, pp. 144–149.

[82] R. Baranowski, S. Di Carlo, N. Hatami, M. E. Imhof, M. A. Kochte, P. Prinetto, *et al.*, "Efficient Multi-Level Fault Simulation of HW/SW Systems for Structural Faults," *Science China Information Sciences*, vol. 54, no. 9, pp. 1784–1796, 2011.

[83] L. Pintard, "From Safety Analysis to Experimental Validation by Fault Injection–Case of Automotive Embedded Systems," PhD thesis, INP Toulouse, 2015.

[84] S. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Application Resiliency Analyzer for Transient Faults," *Micro, IEEE*, vol. 33, no. 3, pp. 58–66, 2013.

[85] L. Berrojo, I. González, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New Techniques for Speeding-Up Fault-Injection Campaigns," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, IEEE, 2002, pp. 847–852.

[86] R. Mariani, *Method for Performing Failure Mode and Effects Analysis of an Integrated Circuit and Computer Program Product Therefore*, US Patent 7,937,679, May 2011. [Online]. Available: `https://www.google.com/patents/US7937679`.

[87] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo, "Fault-List Collapsing for Fault-Injection Experiments," in *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*, IEEE, 1998, pp. 383–388.

[88] P. R. Maier, V. Kleeberger, D. Mueller-Gritschneder, and U. Schlichtmann, "Fault Injection at Host-Compiled Level with Static Fault Set Reduction for SoC Firmware Robustness Testing," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ACM, 2016, p. 18.

[89] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer, 2004.

[90] J. S. Plank, M. Beck, G. Kingsley, and K. Li, *Libckpt: Transparent Checkpointing Under UNIX*. Computer Science Department, 1994.

[91] C. Hernandez and J. Abella, "Low-Cost Checkpointing in Automotive Safety-Relevant Systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, IEEE, 2015, pp. 91–96.

[92] S. Sutherland, "The IEEE Verilog 1364-2001 Standard What's New, and Why You Need It," in *9th Internatioinal HDL Conference (HDLCon)*, 2000.

[93] M. Graphics®, *Questa® SIM User's Manual*, Accessed: 2016-2-23. [Online]. Available: `http://rise.cse.iitm.ac.in/people/faculty/kama/prof/questa_sim_user_manual.pdf`.

[94] Synopsys®, *VCSi User Guide 10.3*, 2005.

[95] Cadence®, *Incisive® Enterprise Simulator User Guide*.

[96] R. Stallman, R. H. Pesch, S. Shebs, *et al.*, *GDB User Manual: Debugging with GDB (The GNU Source-Level Debugger)*, 2014.

[97] C. Artho, K. Suzaki, M. Hagiya, W. Leungwattanakit, R. Potter, E. Platon, Y. Tanabe, F. Weitl, and M. Yamamoto, "Using Checkpointing and Virtualization for Fault Injection," *International Journal of Networking and Computing*, vol. 5, no. 2, pp. 347–372, 2015.

[98] O. S. Initiative *et al.*, "IEEE Standard SystemC Language Reference Manual," *IEEE Computer Society*, 2006.

[99] M. Monton, J. Engblom, C. Schröder, J. Carrabina, and M. Burton, "Checkpoint and Restore for SystemC Models," in *Advances in Design Methods from Modeling Languages for Embedded Systems and SoC's*, Springer, 2010, pp. 41–57.

[100] P. Yang and K. Gopalan, *System and Method for Security and Privacy Aware Virtual Machine Checkpointing*, US Patent App. 14/040,820. Accessed: 2016-04-11, Apr. 2014. [Online]. Available: `https://www.google.com/patents/US20140095821`.

[101] Y. Hollander and Y. Feldman, *Automatic Debug Apparatus and Method for Automatic Debug of an Integrated Circuit Design*, US Patent 8,302,050. Accessed: 2016-03-10, Oct. 2012. [Online]. Available: `https://www.google.com/patents/US8302050`.

[102] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical Fault Injection: Quantified Error and Confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, Apr. 2009, pp. 502–506.

[103] J.-H. Oetjens, O Bringmann, M Chaari, W Ecker, B.-A. Tabacaru, *et al.*, "Safety Evaluation of Automotive Electronics Using Virtual Prototypes: State of the Art and Research Challenges," in *Design Automation Conference (DAC), 51st ACM/EDAC/IEEE*, IEEE, 2014, pp. 1–6.

[104] B.-A. Tabacaru, M. Chaari, W. Ecker, and T. Kruse, "A Meta-Modeling-Based Approach for Automatic Generation of Fault-Injection Processes," *DVCon Europe*, pp. 1–7, 2014.

[105] ——, "Runtime Fault-Injection Tool for Executable SystemC Models," *DVCon India*, pp. 1–7, 2014.

[106] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Comparison of Different Fault-Injection Methods into TLM Models," *Resiliency in Embedded Electronic Systems (REES), 1st International ESWEEK Workshop on*, pp. 1–6, 2015.

[107] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, K. Liu, N. Hatami, C. Novello, H. Post, and A. von Schwerin, "Fault-Injection Techniques for TLM-Based Virtual Prototypes," *Forum on Specification and Design Languages (FDL) – Work in Progress (WiP)*, pp. 1–4, 2015.

[108] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Fault-Effect Analysis on Multiple Abstraction Levels in Hardware Modeling," *DVCon USA*, pp. 1–12, 2016.

[109] ——, "Fault-Effect Analysis on System-Level Hardware Modeling using Virtual Prototypes," *Forum on Specification and Design Languages (FDL)*, pp. 1–7, 2016.

[110] ——, "Speeding up Safety Verification by Fault Abstraction and Simulation to Transaction Level," *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pp. 1–6, 2016.

[111] ——, "Safety-Verification Flow Sporting Gate-Level Accuracy and Near Virtual-Prototype Speed," *DVCon Europe*, pp. 1–8, 2016.

[112] ——, "Gate-Level-Accurate Fault-Effect Analysis at Virtual-Prototype Speed," *ERCIM/EWICS/ARTEMIS Workshop on "Dependable Embedded and Cyber-physical Systems and Systems-of-Systems" (DECSoS'16)*, pp. 1–13, 2016.

[113] ——, "Safety Analysis on Multiple Abstraction Levels," *DATE*, 2016.

[114] ——, "Optimization of Transient-Fault Injection Through Analysis of Simulation Traces," *edaWorkshop*, pp. 1–6, 2016.

[115] ——, "Efficient Checkpointing-Based Safety-Verification Flow Using Compiled-Code Simulation," *Digital System Design (DSD), 2016 Euromicro Conference on*, pp. 1–8, 2016.

[116] M. Chaari, W. Ecker, T. Kruse, and B.-A. Tabacaru, "Automation of Failure Propagation Analysis through Metamodeling and Code Generation," in *ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ)*, 2015.

[117] M. Chaari, B.-A. Tabacaru, W. Ecker, C. Novello, and T. Kruse, "Bridging the Gap Between Probabilistic Safety Analysis and Fault Injection in Virtual Prototypes," in *1st International Workshop on Resiliency in Embedded Electronic Systems, Amsterdam, The Netherlands*, 2015, pp. 34–35.

[118] M. Chaari, W. Ecker, C. Novello, B.-A. Tabacaru, and T. Kruse, "A Model-Based and Simulation-Assisted FMEDA Approach for Safety-Relevant E/E Systems," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015, pp. 1–6.

[119] M. Chaari, W. Ecker, T. Kruse, C. Novello, and B.-A. Tabacaru, "Efficient Exploration of Safety-Relevant Systems Through a Link Between Analysis and Simulation," in *Design and Verification Conference & Exhibition DVCon Europe*, 2016.

[120] M. Chaari, W. Ecker, B.-A. Tabacaru, C. Novello, and T. Kruse, "Linking Model-Based Safety Analysis to Fault Injection and Simulation in Virtual Prototypes," in *Electronic Design Automation Workshop (edaWorkshop 16)*, 2016.

[121] M. Chaari, W. Ecker, and B.-A. Tabacaru, "Towards Cross-Domain and Multi-Level Dependability Analysis Through Metamodeling and Code Generation," in *Electronic Design Automation Workshop (edaWorkshop 16)*, 2016.

[122] M. Chaari, W. Ecker, T. Kruse, C. Novello, and B.-A. Tabacaru, "Transformation of Failure Propagation Models into Fault Trees for Safety Evaluation Purposes," in *Dependable Systems and Networks (DSN) Industrial Track*, 2016.

[123] W. Müller, W. Rosenstiel, and J. Ruf, *SystemC: Methodologies and Applications*. Springer, 2003.

[124] F. Ghenassia *et al.*, *Transaction-Level Modeling with SystemC*. Springer, 2005.

[125] G. Arnout, "SystemC Standard," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ACM, 2000, pp. 573–578.

[126] M.-C. Chiang, T.-C. Yeh, and G.-F. Tseng, "A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 593–606, 2011.

[127] F. Corno, P. Prinetto, M. Rebaudengo, M. S. Reorda, and E. Veiluva, "A Portable ATPG Tool for Parallel and Distributed Systems," in *VLSI Test Symposium, 1995. Proceedings., 13th IEEE*, IEEE, 1995, pp. 29–34.

[128] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, IEEE, 2014, pp. 213–222.

[129] U. V. Methodology, "1.1 User's Guide," *Accellera, May*, 2011.

[130] P. R. Maier, D. Müller-Gritschneder, U. Schlichtmann, and V. B. Kleeberger, "Embedded Software Reliability Testing by Unit-Level Fault Injection," in *21st Asia and South Pacific Design Automation Conference (ASP-DAC 2016)*, 2016.

[131] P. R. Maier, V. Kleeberger, D. Mueller-Gritschneder, and U. Schlichtmann, "Fehlerinjektion auf Unit-Ebene zur Robustheitsverifikation eingebetteter Software," *edaWorkshop*, pp. 1–6, 2016.

[132] M. R. Lyu *et al.*, *Handbook of Software Reliability Engineering*. IEEE computer society press CA, 1996, vol. 222.

[133] L.-T. Wang, N. E. Hoover, E. H. Porter, and J. J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," in *Proceedings of the 24th ACM/IEEE Design Automation Conference*, ACM, 1987, pp. 2–8.

[134] S. Gai, F. Somenzi, and E. Ulrich, "Advances in Concurrent Multilevel Simulation," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 6, no. 6, pp. 1006–1012, 1987.

[135] D. Alexandrescu and E. Costenaro, "Towards Optimized Functional Evaluation of SEE-Induced Failures in Complex Designs," in *2012 IEEE 18th International On-Line Testing Symposium (IOLTS)*, IEEE, 2012, pp. 182–187.

[136] J. B. Gosling, *Simulation in the design of digital electronic systems*. Cambridge University Press, 1993.

[137] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in sub-0.1 μm MOSFET's: A 3-D "Atomistic" Simulation Study," *Electron Devices, IEEE Transactions on*, vol. 45, no. 12, pp. 2505–2513, 1998.

[138] J. Shah, *Hot Carriers in Semiconductor Nanostructures: Physics and Applications*. Elsevier, 2012.

[139] T. Grasser, *Bias Temperature Instability for Devices and Circuits*. Springer Science & Business Media, 2013.

[140] J. Keane and C. H. Kim, *Transistor Aging*, Accessed on 03.01.2017, 15:44, 2011. [Online]. Available: `https://spectrum.ieee.org/semiconductors/processors/transistor-aging/0`.

[141] L. Dirk, M. E. Nelson, J. F. Ziegler, A. Thompson, and T. H. Zabel, "Terrestrial Thermal Neutrons," *Nuclear Science, IEEE Transactions on*, vol. 50, no. 6, pp. 2060–2064, 2003.

[142] M. Tehranipoor, K. Peng, and K. Chakrabarty, *Test and Diagnosis for Small-Delay Defects*. Springer-Verlag New York, 2012.

[143] F. J. Ferguson and T. Larrabee, *Test Pattern Generation for Realistic Bridge Faults in CMOS ICs*. University of California, Santa Cruz, Computer Research Laboratory, 1991.

[144] W.-Y. Chen, S. K. Gupta, and M. A. Breuer, "Test Generation for Crosstalk-Induced Delay in Integrated Circuits," in *Test Conference, 1999. Proceedings. International*, IEEE, 1999, pp. 191–200.

[145] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," *Volume 3A: System Programming Guide, Part 1*, 2010.

[146] ——, "Intel® 64 and IA-32 Architectures Software Developer's Manual," *Volume 3B: System Programming Guide, Part 2*, 2013.

[147] A. L. Silburt, A. Evans, I. Perryman, S.-J. Wen, and D. Alexandrescu, "Design for Soft Error Resiliency in Internet Core Routers," *IEEE Transactions on Nuclear Science*, vol. 56, no. 6, pp. 3551–3555, 2009.

[148] W. Snyder, "Verilator and SystemPerl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[149] G. Brat, D. Bushnell, M. Davies, D. Giannakopoulou, F. Howar, and T. Kahsai, "Verifying the Safety of a Flight-Critical System," in *FM 2015: Formal Methods*, Springer, 2015, pp. 308–324.

[150] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards Formal Approaches to System Resilience," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, IEEE, 2013, pp. 41–50.

[151] B. B. Gerstman, *StatPrimer*, Accessed on 06.01.2017, 09:53, 2003. [Online]. Available: `http://www.sjsu.edu/faculty/gerstman/StatPrimer/t-table.pdf`.

[152] *GNU Make: Parallel Execution*, Accessed on 25.01.2017, 12:34. [Online]. Available: https://www.gnu.org/software/make/manual/html_node/Parallel.html.

[153] S. Gauria, *Verilog-VCD File Parser v1.07*, Accessed: 2016-03-10. [Online]. Available: https://pypi.python.org/pypi/Verilog_VCD.

[154] G. M. Amdahl, *Validity Of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, Accessed on 13.01.2017, 15:44, 1967. [Online]. Available: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.

[155] Synopsys®, *Design Compiler® User Guide*, Accessed: 2016-2-23. [Online]. Available: http://acsweb.ucsd.edu/~coz004/DC_user_guide.pdf.

[156] S. Teran and J. Simsic, *8051 Core*, Accessed: 2016-03-14, 2002. [Online]. Available: http://opencores.org/project,8051.

[157] Open Cores, *AltOr32*, Accessed: 2016-03-14, 2012. [Online]. Available: http://opencores.org/project,altor32.

[158] D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[159] T. Hoefler and R. Belli, "Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, ACM, 2015, p. 73.

[160] B. Spanfelner, D. Richter, S. Ebel, U. Wilhelm, W. Branz, and C. Patz, *Challenges in Applying the ISO 26262 for Driver Assistance Systems*, Accessed on 06.06.2016, 14:43. [Online]. Available: http://www.ftm.mw.tum.de/uploads/media/28_Spanfelner.pdf.