







Article

Automated Identification of Application-Dependent Safe Faults in Automotive Systems-on-a-Chips

Ahmet Cagri Bagbaba ^{1,2,*} , Felipe Augusto da Silva ^{1,3} , Matteo Sonza Reorda ⁴ , Said Hamdioui ³ , Maksim Jenihhin ²  and Christian Sauer ¹ 

¹ Cadence Design Systems, 85622 Munich, Germany; dasilva@cadence.com (F.A.d.S.); sauer@cadence.com (C.S.)

² School of Information Technologies, Department of Computer Systems, Tallinn University of Technology, 19086 Tallinn, Estonia; maksim.jenihhin@taltech.ee

³ Mathematics and Computer Science, Department of Quantum and Computer Engineering, Faculty of Electrical Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands; s.hamdioui@tudelft.nl

⁴ Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy; matteo.sonzareorda@polito.it

* Correspondence: abagbaba@cadence.com

Abstract: ISO 26262 requires classifying random hardware faults based on their effects (safe, detected, or undetected) within integrated circuits used in automobiles. In general, this classification is addressed using expert judgment and a combination of tools. However, the growth of integrated circuit complexity creates a huge fault space; hence, this form of fault classification is error prone and time consuming. Therefore, an automated and systematic approach is needed to target hardware fault classification in automotive systems on chips (SoCs), considering the application software. This work focuses on identifying safe faults: the proposed approach utilizes coverage analysis to identify candidate safe faults considering all the constraints coming from the application. Then, the behavior of the application software is modeled so that we can resort to a formal analysis tool. The proposed technique is evaluated on the AutoSoC benchmark running a cruise control application. Resorting to our approach, we could classify 20%, 11%, and 13% of all faults in the central processing unit (CPU), universal asynchronous receiver-transmitter (UART), and controller area network (CAN) as safe faults, respectively. We also show that this classification can increase the diagnostic coverage of software test libraries targeting the CPU and CAN modules by 4% to 6%, increasing the achieved testable fault coverage.

Keywords: automotive systems; fault classification; fault injection; formal methods; functional safety; diagnostic coverage; ISO 26262; safe faults



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

Academic Editor: Patrick Siarry

Received: 18 December 2021

Accepted: 17 January 2022

Published: 20 January 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Complex hardware and software systems are frequently used in safety critical environments such as automobiles, planes, or medical devices. Safety standards have been introduced to estimate and reduce the risk of critical failures in embedded systems utilized in these areas. This risk might correspond to physical injury or damage to the overall health of humans. Therefore, special solutions for hazards mitigation are required to develop systems working in critical domains. Industries in the above domains need to comply with standards focusing on the development of hardware/software components according to system requirements [1]. Concerning the automotive industry, the number of systems on chip (SoCs) and applications deployed in automobiles is significantly increasing with the final objective of developing self-driving cars. Modern automobiles already incorporate more than 100 electronic control units (ECUs) [2] to cope with the challenges originating from complex applications, such as advanced driver assistance systems (ADAS). Complexities of the hardware and software applications escalate on both the architectural and

functional levels. The hardware complexity is defined as how many components and blocks are integrated on a single SoC/chip. The software complexity is related to the number and time complexity of the pieces that should be combined to deliver the functionality and internal interactions. Moreover, migrating to more advanced integrated circuit (IC) technologies poses a more significant challenge for the safety of automobiles since several phenomena, such as nanoelectronics aging, process variation, or electrostatic discharge used in advanced nodes, introduce numerous vulnerabilities [3]. Consequently, the automotive industry has developed the ISO 26262 Road Vehicles Functional Safety Standard [4] to minimize the risks connected to electric and/or electronic systems used in vehicles. For automotive applications, each electronic system must detect and correctly manage a high percentage of potential faults during the operation in the field to avoid life-critical situations. In order to decide which faults could disturb the safety critical functionality of an IC, faults must be classified based on their effects in the operation mode using expert judgment and a combination of tools. From this perspective, faults can be classified as safe or dangerous. A safe fault does not contribute to the violation of the safety goal, whereas a dangerous fault may lead to a failure relevant for the overall system, that is, create a hazard. We note that all the terms and definitions are given in the context of functional safety verification guided by ISO 26262. Examples of safe faults include faults located in parts of an IC that are not used by the application and faults masked by some safety mechanism. Fault classification is of prime importance for the test of ICs in the operational mode. This test can be performed resorting to different solutions, including design for testability (e.g., BIST) and software test libraries (STLs) based on the software-based self-test (SBST) paradigm [5]. In both cases, the identification of safe faults is vital since it enables us to remove safe faults from the initial (normally huge) fault list and to focus the test efforts toward the remaining faults, i.e., the testable ones [6]. Identifying safe faults thus makes it easier to reach the target diagnostic coverage (DC), helping to achieve safety requirements, such as a higher automotive safety integrity level (ASIL) [4]. For these reasons, there is a high demand for an automated, systematic, and comprehensive safe fault identification technique.

The effects of a fault classification flow are summarized in Figure 1, referring to a generic case study. We assume that an SoC runs a single software (SW) application during its operational life and uses an STL as a safety mechanism. Therefore, the DC of this STL must be calculated to prove that it detects dangerous faults up to a certain extent in the target design. In the first step of the flow, without any classification, all the faults are unknown, as shown in Figure 1. Then, an initial classification is performed to identify the first group of structurally safe faults, i.e., those which are safe due to the IC structure (e.g., faults located on lines which are not connected to the IC primary inputs and/or outputs). These kinds of safe faults can be identified using any automatic-test-pattern-generation (ATPG) or formal analysis tool. However, other safe faults may exist, which cannot be identified by these tools; therefore, a considerable amount of faults are still unknown after the first step. The unknown faults need to be further analyzed to check whether their effects may impact the safety critical functionalities or not. Thus, fault simulation with an STL is deployed to classify faults better. In practice, this step (named unoptimized classification in Figure 1) produces inaccurate results since it is often impossible to exhaustively evaluate all possible input stimuli or activate all possible operating modes in an application or system [7]. Undetected faults may correspond either to safe or dangerous faults. As in Figure 1, fault simulation targets unknown faults and classifies them as either detected or undetected based on the propagation of faults. A non-negligible amount of undetected faults may be observed depending on the workload that runs on the target design. Usually, all the undetected faults are pessimistically classified as dangerous. For this reason, the gathered figures from fault simulation may not be representative of the design operational behavior, as not all faults can be accurately classified. DC is calculated in this step using (1), where Detected is the number of faults classified as detected and dangerous by fault simulation; Total is the

size of the target system's fault list; and Safe is the number of safe faults. The purpose is to check if the collected results from fault simulation satisfy the desired safety metrics. If DC is not enough, the test must be improved, or an additional classification effort targeting undetected faults, i.e., a subset of undetected faults, is required to classify their effects. Experts usually perform this step based on their design knowledge; however, this is error prone and time consuming. Consequently, the unoptimized classification implies that there is still room for improvement in the fault classification pessimism. Finally, using the technique presented in this work, a formal analysis approach optimizes the fault classification (named optimized classification) as shown in the fourth bar of Figure 1, which targets the identification of more safe faults, reducing the number of undetected faults and, therefore, the overall pessimism of the classification. The optimized classification decreases the denominator of (1) by classifying more safe faults than in the unoptimized classification, and the DC is increased.

$$DC = Detected / (Total - Safe) \quad (1)$$

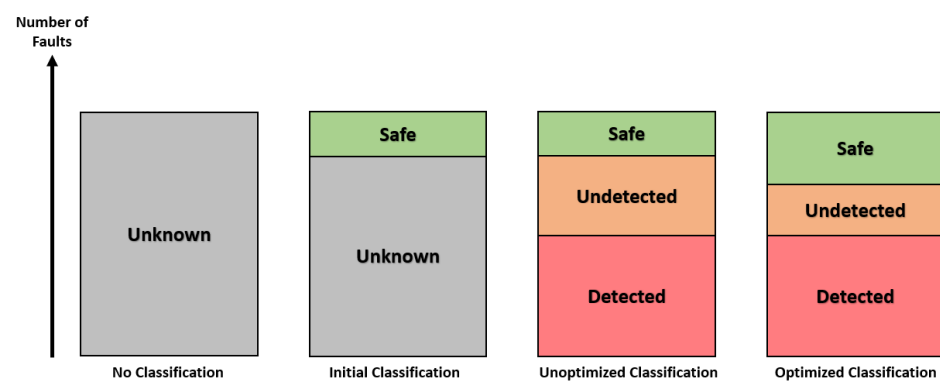


Figure 1. Hardware fault classification flow.

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

As a case study, the AutoSoC benchmark [8], an automotive representative SoC, and the cruise-control-application (CCA) as a target SW application are used. We focused on the CPU core and several peripherals, i.e., the universal asynchronous receiver-transmitter IP (UART) and the controller area network controller IP [9] (hereinafter referred to as CAN).

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

technologies that address the problem. The main contributions of this work can be listed and summarized as follows:

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

The rest of this paper is structured as follows. Section 2 summarizes the previous and related works in the area. Section 3 provides some background, covering hardware fault classification and the techniques to achieve this classification, such as fault simulation and formal analysis. Section 4 defines the App-Safe faults in detail and presents the proposed method step by step. Section 5 briefly describes the AutoSoC benchmark suite, including its CPU, peripherals, and the software application that we use in this work. Section 6 reports and discusses the experimental results of the proposed technique. Finally, Section 7 draws some conclusions.

2. Related Works

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

Several works have explored fault simulation targeting fault classification. For example, Ref. [10] optimizes fault simulation by integrating it into the design verification environment and using the clustering approach to accelerate the fault simulation campaigns. However, using only fault simulation for fault classification is computationally expensive and incomplete; hence it requires additional methods to classify undetected faults. Similarly, Ref. [11] relies only on fault simulation to classify the faults, but there was no additional classification technique proposed. Similarly, Refs. [12–15] deploy fault simulation to classify faults in automotive systems considering the requirements of ISO 26262. In short, when fault simulation is used alone to classify faults, additional techniques targeting the classification of undetected faults are necessary.

Hence, some other works have investigated formal analysis, focusing on safe fault identification. Refs. [16–18] use the ability of the formal techniques to analyze the design behavior. Safe fault identification is also applied to GPUs. For example, ref. [19] employs formal analysis to increase fault coverage when the identification technique is applied to an open-source GPU. These works specifically focused on identifying structurally safe faults, i.e., faults for which there are no test or input stimuli due to the hardware structure, independently of the software and the application.

Researchers have also combined fault simulation and formal analysis leveraging fault classification. Refs. [20,21] have an eclectic approach that makes use of the strength of

different technologies. Even though these works are promising in terms of the results, they still require many manual efforts based on the engineer's expertise.

On the other side, ATPG is also a promising technique to identify safe faults. Examples of this approach are [22–24], which aim at identifying untestable faults in sequential circuits. We note that untestable faults are, by definition, safe faults [25]. In addition, Refs. [6,25,26] resort to ATPG to identify application-dependent safe faults, which is the same target of the work described in this paper. Even though these works can identify safe faults using the ATPG, they still have a manual part in their flow, i.e., they are semi-automated.

Considering application-dependent safe faults, some works have proposed solutions for the classification of these kinds of faults. For example, Ref. [27] explores the use of safe faults to optimize STL fault coverage in microprocessors, which is not safety critical. However, the scope of the work is limited only to CPU modules, and the deployed tests are not automotive representative. Additionally, Ref. [28] focuses on safe fault identification in only CAN; thus, the analysis of a complete automotive representative SoC is missing. In addition, Ref. [28] analyzes a combination of test programs developed for CAN, which makes it weaker as this work examines safe faults when an SoC runs a practical industry-scale software application. Last, the presented work in this paper has a more advanced approach in the sense that the proposed technique is more automated and systematic; hence, it is less error prone and time consuming.

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

3. Background

This section, first, provides basics about hardware fault classification. Then, fault simulation and formal methods for hardware fault classification are explained.

3.1. Hardware Fault Classification

ISO 26262 divides the malfunction of electrical/electronic components into two categories, corresponding to systematic and random faults [4]. A systematic fault is manifested in a deterministic way and can only be prevented by applying process or design measures. On the other hand, a random fault can occur unpredictably during the lifetime of a hardware element. When we consider safety critical designs, such as automotive, medical, or aerospace designs, safety and verification engineers must prove that both the correct and safe functionalities of these designs are guaranteed, taking into account both systematic and random faults.

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

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

- **Safe:** A safe fault does not disturb any safety critical functionality because it is not in safety relevant logic, or it is in safety relevant logic but is unable to impact the safety critical functionality of a design (i.e., it cannot violate a safety goal).

- **Dangerous:** A dangerous fault impacts the safety of the device and creates a hazard that may produce a safety goal violation.

3.2. Fault Simulation

As an integral part of the safety critical IC development, fault simulation is a widely used technique to identify fault effects [29]. Fault simulation tools analyze an RTL or gate-level abstraction of an IC by performing a simulation with some given test stimuli. In general, the fault injection flow is based on the comparison between the results of the good run and those of the faulty run. First, the good run is run to generate reference values. In this step, observation points where the propagation of faults is monitored are specified. Then, the faulty run is executed with faults injected. In the end, the reference values obtained by the good run and the faulty values generated by the faulty run are compared for the classification of each injected fault, and we can determine whether each injected fault is detected or undetected. Faults are classified as detected when at least one output value changes for a specified observation point between the good run and the faulty run. Otherwise, the fault is classified as undetected.

Although fault simulation is a widely used and adopted technique by both industry and academia, it suffers from two problems [7]:

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

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

The following subsection explains how formal methods classify faults, distinguishing between safe and dangerous.

3.3. Formal Methods

Formal methods help to classify faults based on their effects. An analysis is performed to determine whether or not a target design satisfies a set of properties or conditions. This approach is usually a combination of different techniques that employ static analysis and algorithmic calculations. Compared to fault simulation that applies one single stimulus, formal analysis is less limited since it abstracts from any specific stimulus. On the other hand, the computational complexity may limit the formal analysis applicability [28]. In this case, the classification of all faults can be impossible; thus, a formal analysis tool should be fed by formal properties, developed carefully, considering the constraints from the SW application and looking for a compromise between computational feasibility and result accuracy, as it is done in this work.

In general, formal tools apply two checks, structural analysis check and formal analysis check to identify safe faults, as explained below.

3.3.1. Structural Analysis Check Types

In the structural analysis check, formal tools use the topological characteristics of a design to determine the testability of each fault. There are three methods of structural fault analysis:

- Out-of-cone of influence (COI) analysis: This method checks whether a given node is outside the COI of a given observation point(s); in that case, the fault is safe. In Figure 2, all faults located on nodes in the COI of out_1 (shown in green) are safe since the considered observation point is out_0 in the example analysis. It is obvious that stuck-at faults on the cell ports of G3 cannot propagate to out_0 , as they have no physical connection with out_0 . Hence, faults on G3 are safe.
- Unactivatable analysis: This is to check if a SA0 or SA1 fault is located on a node that is constant 0 or 1; if so, the fault cannot be activated. In this case, the fault is unactivatable and safe. In Figure 3, assuming that in_0 is tied to logic zero, f_0 for SA0 is unactivatable and safe.
- Unpropagatable analysis: This is performed to check if a fault is activated and in the COI of the considered observation point but cannot be propagated to the outputs. In this case, the fault is safe. In Figure 3, the AND gate G2 can block the propagation of f_1 if one of the in_1 or in_2 is always set with the logic value zero. Hence, f_1 would be safe for SA1 or SA0, as it can never be propagated to out_0 .

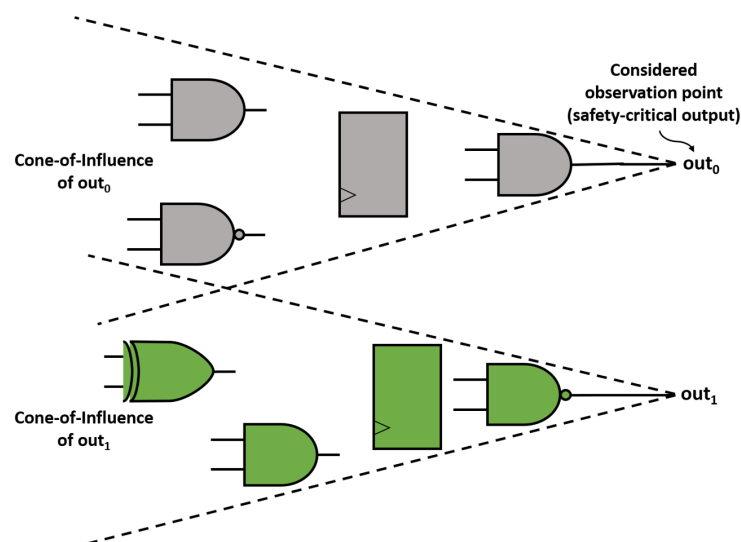


Figure 2. Out-of-COI example when out_0 is the only safety critical output.

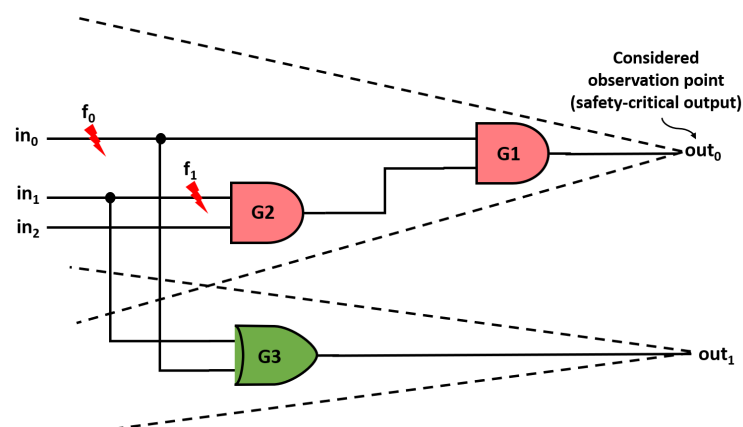


Figure 3. Unactivatable and unpropagatable analysis example.

3.3.2. Formal Analysis Check Types

As opposed to structural analysis checks for which physical connections of a design are taken into account, formal analysis checks are used to classify faults as well. The approach uses a good machine and bad machine similar to the fault simulation and injects a fault in the bad machine for formal analysis. In the end, the output signal values of good

and bad machines are compared to check whether an injected fault is propagated or not. A formal tool generally generates a Boolean representation of the function implemented by the circuit (or part of it) and uses formal techniques as explained above to prove this Boolean equation. Formal analysis tools use various engines based on Boolean expressions representation and manipulation techniques, such as binary decision diagrams (BDDs) [30] to prove the formal properties exhaustively. There are two types of this analysis:

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

The technique described in Section 4 deploys both structural and formal analysis checks resorting to formal methods.

4. Proposed Application-Dependent Safe Fault Identification Method

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

In Section 3, we explain that a safe fault does not disturb any safety critical functionality because it is not located in any safety relevant logic or is in a safety relevant component. Based on this explanation, we further classify safe faults as follows:

- Structurally safe (Str-Safe): These are faults that cannot be activated or propagated to the outputs of interest by any test sequence because of the design's structural constraints. For example, a fault in the redundant logic or a floating net (i.e., any net that does not have a load) is Str-Safe. Another example is supply0 and supply1 nets. Specifically, a SA0 fault on supply0 net and a SA1 fault on a supply1 net are Str-Safe. Finally, a SA1 fault on a pull-up gate and a SA0 fault on a pull-down gate are Str-Safe.
- Functionally safe: As opposed to structurally safe faults, a test or test sequence for functionally safe faults exists, and their effects may propagate to design outputs. However, they do not affect any safety critical functionality. For example, faults in the debug unit of a CPU not used due to hardware configuration are functionally safe.

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

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

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

4.1. The Proposed Flow

In Figure 4, the proposed flow to identify App-Safe faults is shown, step by step. At the beginning of the flow, we have a design-under-test (DUT) circuit (typically, an

SoC) and a SW application running on it. First, we run several logic simulations with different representative input data sets. The goal of running logic simulations is to analyze the design's behavior when it runs the SW application. Next, application-specific formal properties are developed to translate the design's operational behavior into the formal environment. Formal properties provide input to the formal analysis tool to identify App-Safe faults. Finally, the formal analysis tool is deployed, and safe faults are listed. In the following subsections, we discuss each step in detail.

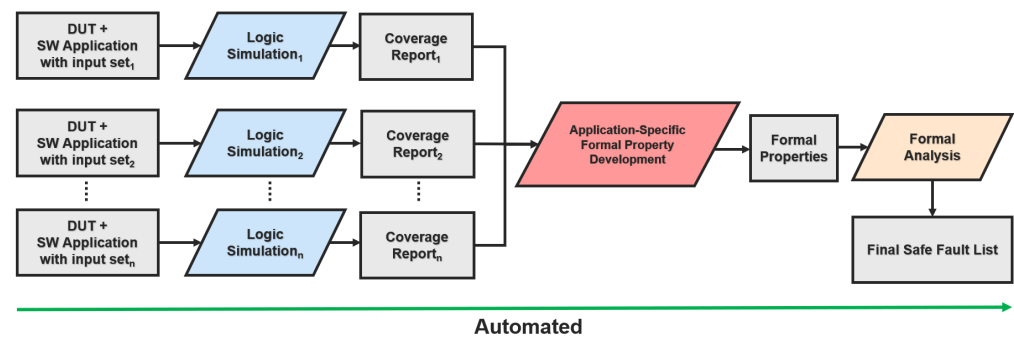


Figure 4. Proposed application-dependent safe fault identification flow.

4.1.1. Logic Simulation

In this step, we perform several logic simulations on the design under test (DUT) executing the SW application with different representative realistic data input sequences, i.e., set_1 to set_n , as shown in Figure 4. More than one logic simulation is performed when each of them runs with different input data since we aim to identify which design parts are independent of the input data set. The purpose of performing logic simulations is two-fold:

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

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

In general, logic simulations aim to detect which points are not toggled, as these are App-Safe candidates that must be addressed. Concerning coverage metrics, the proposed work focuses on hardware code coverage that assesses how well the stimuli exercise the design code by pointing to design components that did not meet the desired coverage criteria [31]. Our technique deploys toggle and block coverage sub-types of design code coverage to identify App-Safe faults. Block coverage is a primary code coverage metric that identifies which lines in the code have been executed and which have not. On the other hand, toggle coverage monitors, collects, and reports the signal toggle activity, allowing the identification of unused signals or signals that remain at a constant value of 0 or 1.

The block and toggle coverage metrics provide insight into the SW application behavior during the operational life of an IC. Thus, we can identify App-Safe candidates included in the functionally safe fault list. More specifically, block coverage can indicate that some states are never activated, indicating that the SW application does not use the corresponding design components. Likewise, constant signals identified by toggle coverage can highlight invalid configurations, not utilized functions, among others. Moreover, the combination of block and toggle coverage data should be carefully analyzed because they can point out further information about the SW application's behavior. For the sake of an example, an untoggled signal may never activate a state machine block, and this can cause some other blocks to remain unactivated during the simulation. The small Verilog code in Listing 1 and Table 1 illustrates block coverage, toggle coverage and explains why both of them should be carefully analyzed. Listing 1 shows that *rf_data_in* block is never activated since *break_error* is never toggled to logic 1 as shown in Table 1. This coverage results also means that *rf_data_in* never gets the right-hand side value at line 402, as the block is not

activated. This example points out the importance of assessing block and toggle coverage together.

After running logic simulations and measuring the hardware code coverage metrics presented above, the hardware coverage metrics data are available for report generation and analysis. At the end of this step, coverage reports represent the design's operational behavior under the effect of different input data sets. When this behavior is translated into formal properties, as explained in the following subsection, we call them application-specific formal properties.

Listing 1. Block coverage example: *rf_data_in* is not executed.

```

if(srx_pad_i | break_error)
// The following "begin" block is covered (100%)
begin
    if(break_error)
        // The following block is not covered (0%)
        rf_data_in <= {8'b0, 3'b100};
    else
        // The following block is covered (100%)
        rf_data_in <= {rshift, 1'b0, rparity_error, rframing_error};
        // The following block is covered (100%)
        rf_push <= 1'b1;
        rstate <= sr_idle;
    end
end

```

Table 1. Toggle coverage example: *break_error* is not toggled.

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

4.1.2. Application-Specific Formal Property Development

The development cycle of ICs begins with inferring the specification and requirements of the target system. Additionally, the DUT must be verified with a formulated verification plan, which is defined by both design and verification engineers. Then, features or requirements of the DUT are created and mapped to the formal properties to deploy them in a formal analysis tool [32]. Formal properties are created from the design specification and implementation decisions. Thus, after extracting the target system's operational behavior through logic simulations, in this step, we translate this behavior to the formal properties to be used in a formal analysis tool, which will identify additional App-safe faults.

We use two types of formal properties to define the correct behavior of the design. The first one is assume statement, which creates an assumption for the specified Boolean expression that evaluates to either true or false. In the general sense, it specifies that the given property is an assumption and is used to generate the input stimulus. Hence, assume statements can be helpful when we define a design configuration or to inform the tool how the design inputs can behave. Without this assumption, a formal tool checks all possible input combinations of the DUT. There are two benefits of using assume statements in the formal environment. First, it allows excluding illegal input combinations when known. Legal inputs are those that we expect to see during normal operation. It is not realistic to expect the design to behave correctly when all possible input combinations are being applied, unless we explicitly define every possible set of input combinations that the design can theoretically see. The second benefit of using assume statements is that it intentionally reduces the state space, which is exhaustive when no assumption is defined. For example, as we want to prepare our formal environment considering the design's operational behavior, we should disable the *scan_enable* pin, as the scan chain is not activated during the operation and is used only for test purposes. In this case, the assume statement given in (2) is created to inform the formal tool about the *scan_enable*

signal behavior; thus, the input test stimuli of a formal analysis tool are limited accordingly. The command given in (2) simply informs the tool that *scan_enable* is always logic-0. Assume statements also increase the safe fault identification capacity of a formal analysis tool by guiding it. Moreover, similar to the example given below, the input ports of design instances are suitable candidates for assume statements.

$$\text{assume} - \text{env} \{ \text{scan_enable} == 1'b0 \} \quad (2)$$

The second formal property is the fault propagation barrier, which creates a formal barrier that blocks the propagation of a fault. In this case, faults cannot propagate after this barrier; therefore, they cannot disturb any safety critical functionality. For instance, knowing that the debug unit is not used in the design's operational mode, we can block all faults to propagate from it and identify more App-Safe faults. As seen in (3), the formal analysis tool is asked to block all faults propagated to *du_dat_o*, which is the debug unit's data output signal. As in this given example, output ports are proper candidates for a fault propagation barrier as opposed to assume statements, for which input ports are suitable candidates.

$$\text{check_fsv} - \text{barrier} \{ \text{du_dat_o} \} \quad (3)$$

Consequently, the application-specific formal properties [33] can be developed using assume statements and fault propagation barriers. By doing so, the internal architecture and logical details of the target system, the operational constraints (if any), or the initial configuration of the design can be defined as formal properties to be used in the formal analysis step. Therefore, the design's operational behavior can be transferred from the logic simulations into the formal analysis tool. The following subsection explains how the formal analysis tool uses these application-specific formal properties.

4.1.3. Formal Analysis

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

The formal analysis flow, which includes three phases, is shown in Figure 5. Phase I begins with the creation of input files that are the formal properties established in the previous stage and the DUT. Then, it continues with the development of the tool command language (TCL) setup script for the formal analysis tool. The setup script consists of Verilog files, libraries, and formal property files. The setup script first analyzes design and property files to check for syntax errors. Then, it defines clock and reset signals. The clock definition is to specify the characteristics of how the clock is driven during a formal analysis run. The reset specification aims at bringing the design to a known state and avoiding unreachable failure states. In the next step, warnings are generated by the formal analysis tool if there is a mismatch between formal properties and the DUT. For example, a signal tied to the ground in the DUT and the assume statement that defines this signal as if it is always logic-1 can create a mismatch, and a warning is generated. However, as we automatically translate coverage reports to the formal properties, this is not the case for the work proposed in this work. Then, in Phase II, the formal engine proves the formal properties by running the structural and formal checks as presented in Section 3.3. Finally, in Phase III, App-Safe faults are identified and reported.

In brief, a formal analysis tool uses formal properties to generate safe faults. When we include formal properties driven by SW application, as mentioned before, we enable

the tool to work in a well-specified configuration. Hence, formal analysis with the formal properties increases the number of identified App-Safe faults.

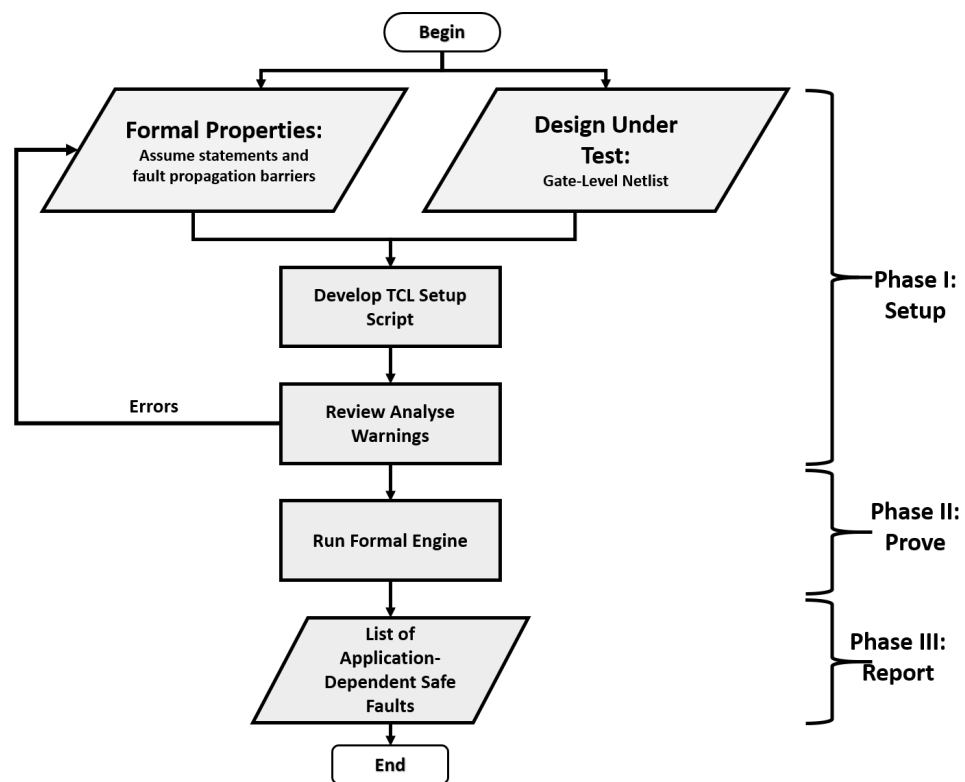


Figure 5. Formal analysis phases.

5. Case Study: The AutoSoC Benchmark Suite

The proposed application-dependent safe fault identification method is evaluated on the AutoSoC benchmark suite, which we conceptualized in [8]. The AutoSoC is an open-source benchmark suite, incorporating all required elements in the format of a configurable SoC. It is developed to support research in the automotive domain by providing varied hardware configurations, safety mechanisms, and representative software applications. In this section, we explain the AutoSoC by detailing its CPU and other functional blocks.

5.1. General Architecture of the AutoSoC

Developed by characterization of commercial CPUs used in the automotive field, the AutoSoC has two main processing units as the safety island and the application specific block, as illustrated in Figure 6. While the safety island handles all safety-critical processes driven by ISO 26262 [4], the application-specific block executes the hardware needed for application-specific processing. It is also important to note that the safety island and the application-specific block have dedicated software stacks to execute distinguished applications. The interconnect block deploys Wishbone Bus for internal SoC communication. Additionally, the remaining blocks in Figure 6 are included to fulfill the requirements for communication, security, and general infrastructure.

Since the AutoSoC is implemented as a modular, it has several configurations as detailed in [8]. One of these configurations named *AutoSoC QM* is used in this work. This configuration is a fully functional version of the benchmark suite. When considering functional blocks of the AutoSoC shown in Figure 6, the *AutoSoC QM* configuration has only the application-specific block, but here, the presented work remains suitable to all available configurations of the AutoSoC.

The main CPU used in the development of the AutoSoC is the *mor1kx* implementation of the OpenRISC [34]. This implementation provides all necessary tools and examples

for developing SoCs, such as CPU, memory, debug unit, communication protocols, and a bus. Concerning software resources, the AutoSoC includes several options, some of which come from the *mor1kx* package and the others developed by ourselves in conformity with automotive functional safety analysis. These software resources are available as both BareMetal, and the real-time executive for multiprocessor systems (RTEMS) real-time operating system [35]. Furthermore, the automotive cruise control application (CCA) is developed and targeted for safe fault identification. This application is based on BareMetal and the RTEMS operational system and covers several tasks: reading vehicle sensor data from UART and CAN, computing actuation, and setting engine parameters. In addition, the AutoSoC has STL programs that target the CPU (*mor1kx_cpu*). These STL programs are developed for online testing of the AutoSoC. The current available STL presented in the open-source AutoSoC package comprises 16 test programs [8].

Finally, the AutoSoC is available at both the register-transfer level (RT-Level) and gate level. The synthesis is performed using Cadence GPDK045 (45nm CMOS Generic Process Design Kits). The proposed approach in this paper is demonstrated using the gate-level model of the AutoSoC.

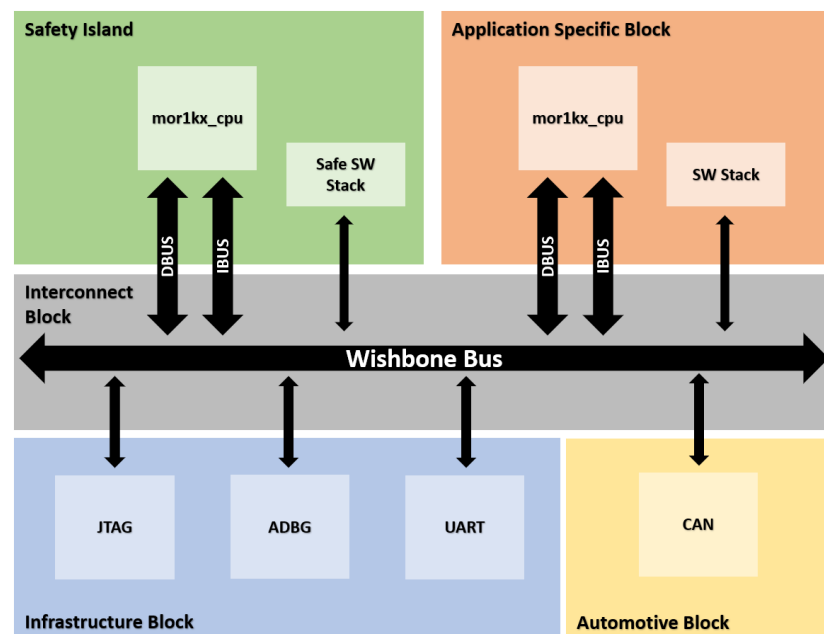


Figure 6. AutoSoC functional blocks.

5.2. UART IP

The AutoSoC benchmark suite includes a UART IP, which incorporates the industry standard National Semiconductors' 16550A device features. Furthermore, as it is a well-known and widely-used communication standard by the industry and academia, and the proposed safe fault identification method is also extended to the UART. In this subsection, we provide details about the adopted core.

UART is a block of circuitry that uses asynchronous serial communication with configurable speed. It operates data transfer by receiving data from a peripheral device or a CPU. Moreover, the UART includes an interrupt system and control capability tailored to minimize software management of the communication link. The UART IP used in the AutoSoC operates in a 32-bit bus mode fully compatible with Wishbone Bus. As depicted in Figure 7, the UART core consists of receive logic, control, and status registers, modem control module, transmit logic, Baud generator logic, and interrupt logic. Incoming serial messages are received by the RX shift register, whose Baud rate is programmable through Baud generator logic. Received messages are placed in the receive FIFO if the incoming messages have no problems. On the contrary, the TX shift register handles the transmission of data written to the transmit FIFO. Control and status registers allow the specification

and observation of the format of the asynchronous data communication used. Modem control has registers that allow transferring control signals to a modem connected to the UART. The UART IP also has Baud generator logic to control transmit and receive data rates. Finally, interrupt logic allows enabling and disabling interrupt generation by the UART.

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

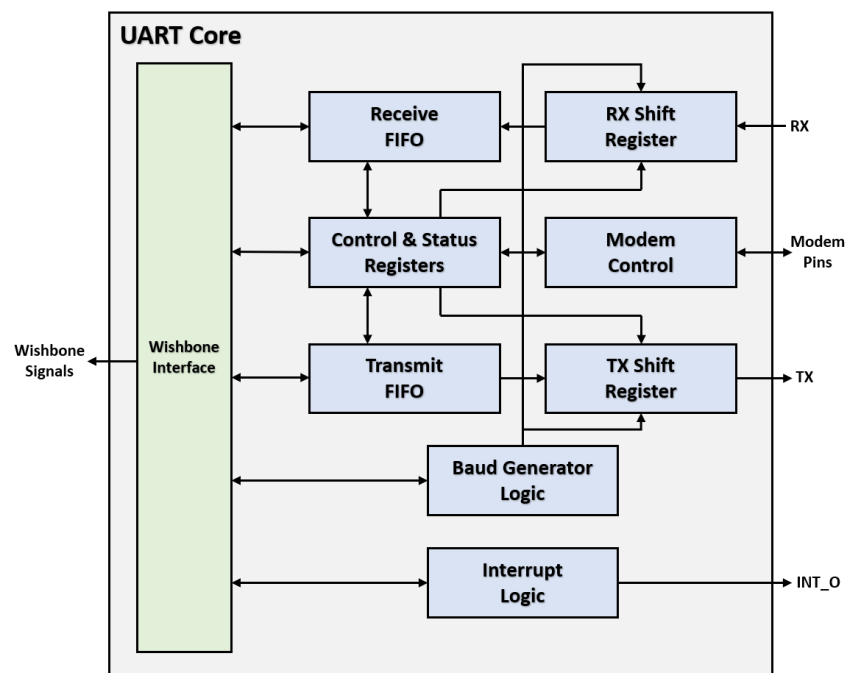


Figure 7. Block diagram of the UART IP.

5.3. CAN Controller IP

The CAN is a communication bus standard introduced by Bosch in 1986. It is intended to work in the automotive field for serial communication applications among microcontroller units. The CAN has several benefits; it is low-cost, and it has the ability to self-diagnose and repair data errors. These features promote CAN's popularity in the automotive and some other industries, such as medical or aerospace [36]. As it represents the automotive industry's challenges, we validate the proposed safe fault identification method on CAN.

The AutoSoC benchmark suite has open hardware implementation of the SJA1000 [9], which is a standalone controller for the CAN, developed by Philips Semiconductors in the early 2000s. Figure 8 shows the block diagram of SJA1000 CAN. The CAN transceiver is a module to connect other nodes to the CAN. The CAN core block controls the reception and transmission of CAN frames. The interface management logic implements the CAN interface as a link to the host CPU through its set of registers. Additionally, this block configures the operational mode of CAN, whether it works in *BasicCAN* or *PeliCAN* mode. The transmit buffer stores messages in extended or standard format. The CAN core block reads messages from the transmit buffer whenever the interface management logic forces it. The acceptance filter comes into prominence when receiving a message. It checks whether the message on the bus has to be stored by the CAN or not. All received messages accepted by the acceptance filter are stored in the receive FIFO.

As the AutoSoC benchmark suite uses a Wishbone Bus, the adopted CAN is directly connected without the need for bridges between different bus interfaces. When it is required to add another node to be communicated with the Host CPU, the CAN Transceiver provides

a straightforward way for connection. Moreover, the AutoSoC benchmark suite provides an STL for the self-test of the CAN. As it is explained in [37], the developed STLs implement an effective in-field test for the CAN based on a functional approach and also provide experimental evidence to demonstrate its effectiveness.

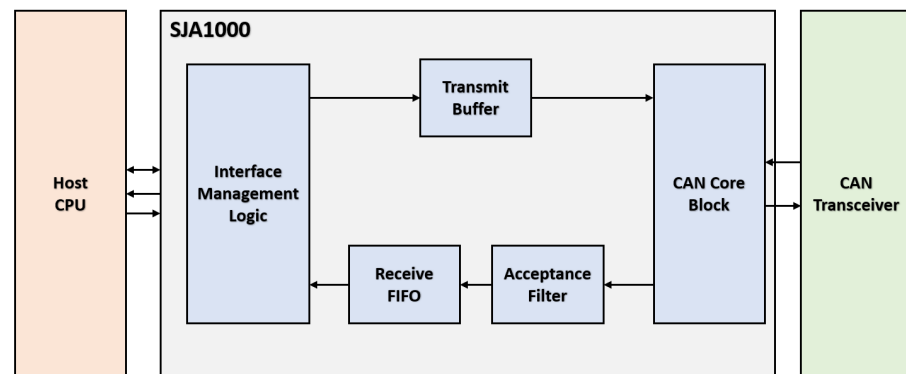


Figure 8. Block diagram of the adopted SJA1000 CAN controller IP.

6. Experimental Setup and Results

This section first describes the experimental setup we used to quantitatively assess the effectiveness of the proposed approach. Then we provide the results in a separate subsection by focusing on CPU, UART, CAN, and finally the combined results.

6.1. Experimental Setup

In order to demonstrate the effectiveness of the proposed App-Safe fault identification method, we used the experimental setup shown in Figure 9. Our setup is composed of two AutoSoC nodes; each includes a CAN and a UART IP to communicate with each other, and one of the two AutoSoCs (named AutoSoC-0) is assumed to be active, whereas the other (named AutoSoC-1) is the passive node. Moreover, CCA accesses CAN or UART in both the AutoSoC-0 and the AutoSoC-1. Thus, each CCA comes in two modes, even though the executed steps are symmetric; the two AutoSoC nodes alternatively receive and send messages in the same configuration. Furthermore, even if it is changeable, AutoSoC-0 receives messages first, while AutoSoC-1 transmits first in our experimental setup. Finally, the whole system is simulated at the gate level.

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

In brief, we first performed logic simulations using the hardware configuration described before, which runs the CCA SW application using different input data sets, as shown in Figure 4. Then, coverage reports are generated and translated into application-specific formal properties that configure the formal analysis environment according to the SW application's behavior. Finally, the formal analysis tool is deployed to identify App-Safe faults.

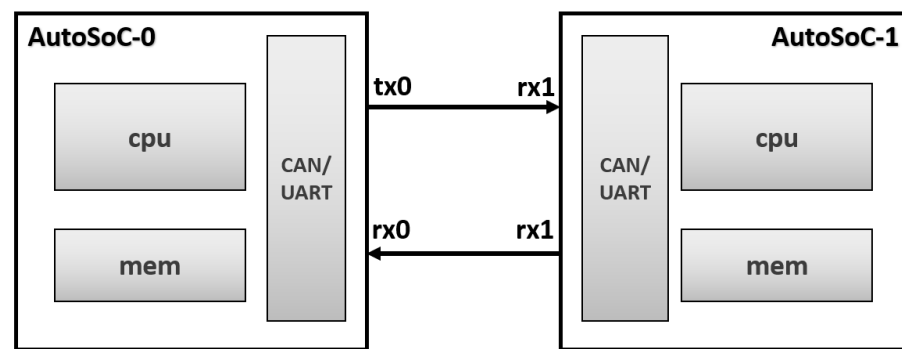


Figure 9. Experimental setup composed of two AutoSoC nodes.

6.2. Experimental Results

This subsection presents the identified safe faults in CPU, UART, and CAN, respectively. Then, the combined results (fault simulation + formal analysis) are reported for the CPU and CAN modules (this step does not include the analysis of UART).

6.2.1. Safe Faults in CPU

Firstly, App-Safe fault identification is checked in the CPU core (which has 96,354 faults in total) when it runs a SW application. We summarize the safe fault results of the CPU in Table 2.

Table 2 categorizes the results based on the analysis we run. In the top row, it can be seen that we performed four analyses as follows:

- **Application-independent:** The formal analysis tool is deployed on the gate-level netlist of the AutoSoC without any formal properties, meaning that the identified safe faults are valid for any SW application.
- **BareMetal-CCA:** The CCA runs BareMetal, which refers to running the SW application directly on a CPU without the support of an operating system. In order to perform this analysis, the gate-level netlist of AutoSoC and the formal properties (as explained in Section 4.1.2) are used as inputs to the formal analysis tool.
- **RTEMS-CCA:** Unlike BareMetal-CCA, the SW application runs on an operating system in this analysis, meaning that it can start and stop different processes concurrently. The RTEMS-CCA causes higher signal activity when compared to BareMetal-CCA, as it runs on operating systems that trigger more signals. In addition, RTEMS-CCA uses two additional opcodes compared to BareMetal-CCA. This means that RTEMS-CCA triggers more design components than BareMetal-CCA.
- **BareMetal-Sum:** For this analysis, we use an entirely different SW application than CCA. The application performs a sum operation, and it has fewer opcodes than BareMetal-CCA. This SW application aims to show how App-Safe faults change when the CPU is running a different application.

In brief, App-Safe faults are originated from what a SW application executes in an IC. For example, some design components are not accessed during the design's operational life, such as debug units or scan chains. In addition, unused opcodes cause App-Safe faults, meaning that if (for example) the multiplication opcode is not used in the SW application that runs on the IC, all signals related to multiplication hardware become App-Safe faults, as they are not exercised. Table 2 reports the results for the CPU core, also detailing the results achieved on each component module inside it. In the application-independent analysis, the formal analysis tool identifies 8.785% safe faults with respect to all faults in the CPU. We highlight that all the identified safe faults in the application-independent analysis are Str-Safe faults because the formal tool could not identify any safe faults using the formal fault analysis check types mentioned in Section 3.3.2 without formal properties.

Concerning the three application-dependent analyses (BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum):

- The *top* module includes connectivity signals and configuration-related signals. Among these, debug unit's address and data signals, interrupt request signals, multicore configuration signals, special-purpose-register signals are identified as safe in all analyses since they are not activated due to the SW applications configuration. Depending on the opcodes used in the applications, there are slight differences in BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum. For example, RTEMS-CCA triggers exception signals, which are connected to the top level.
- The *Decode_execute Unit* is the module where the instruction memory management unit (IMMU) and the data memory management unit (DMMU) signals take part. Many safe faults are identified in the IMMU and DMMU, which are not used by the SW applications. The number of safe faults is different between BareMetal-CCA and RTEMS-CCA because of the exception signals used by RTEMS-CCA, as mentioned above. In addition, the deviation between BareMetal-CCA and BareMetal-Sum is due to division and multiplication-related signals, which BareMetal-Sum does not use.
- The *load-store unit* computes the addresses used by load and store instructions. Safe faults may exist, as not all addresses are used by the SW applications. In addition, some connection signals create a slight difference between BareMetal-CCA and RTEMS-CCA.
- The *fetch stage* fetches the next instruction from memory into the instruction register. Therefore, it is directly associated with the address range, which is not fully covered by the SW application. Therefore, safe faults can be identified in this unit. In addition, the difference between BareMetal-CCA and RTEMS-CCA is due to exception signals.
- The *control stage* has the most considerable impact on the number of identified safe faults. This unit contains features such as a tick timer, interrupts, and configuration registers. Since the CPU configuration is the same in all applications, configuration registers create the same amount of safe faults. However, the tick-timer unit has a higher activity in RTEMS-CCA; hence, it has fewer safe faults when the CPU runs RTEMS-CCA.
- Concerning the *arithmetic logic unit*, the proposed technique identifies the same amount of safe faults in BareMetal-CCA and RTEMS-CCA, as they use the same arithmetic opcodes. However, BareMetal-Sum performs only addition operations; therefore, all the other arithmetic operations contribute to the safe faults.
- The *decode unit* is directly affected by the used opcodes; hence, there is a difference between the numbers of safe faults, as all three analyses use different numbers of opcodes.

The results in Table 2 show that the percentage of safe faults varies widely from one module to another, depending on the tasks performed by the modules. In addition, the number of App-Safe faults is relevant, accounting for about 20%, 14%, and 40% in BareMetal-CCA, RTEMS-CCA, and BareMetal-Sum applications, respectively.

Table 2. Safe faults in CPU.

CPU Modules	Application-Independent		Baremetal-CCA		RTEMS-CCA		Baremetal-Sum	
	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults
Top	1679	1.743%	1725	1.790%	1716	1.781%	1717	1.782%
Register File	2	0.002%	5	0.005%	2	0.002%	5	0.005%
Decode_Execute Unit	651	0.676%	844	0.876%	719	0.746%	949	0.985%
Load Store Unit	910	0.944%	2380	2.470%	2317	2.405%	2380	2.470%
WriteBack Mux Unit	0	0.000%	0	0.000%	0	0.000%	76	0.079%
Fetch Stage	976	1.013%	1230	1.277%	1195	1.240%	1230	1.277%
Control Stage	3966	4.116%	11,618	12.058%	6418	6.661%	11,618	12.058%
Arithmetic Logic Unit	55	0.057%	1000	1.038%	1000	1.038%	19,478	20.215%
Decode Unit	5	0.005%	267	0.277%	16	0.017%	315	0.327%
Branch Prediction Unit	0	0.000%	0	0.000%	0	0.000%	0	0.000%
TOTAL	8465	8.785%	19,484	20.221%	13,670	14.187%	38,193	39.638%

6.2.2. Safe Faults in UART

Concerning the UART module, which has 19,120 faults in total, we followed the same procedure using two scenarios (application-independent, and CCA is compared), and the results are detailed in Table 3. We also noted that there is no difference between BareMetal-CCA or RTEMS-CCA, so we only report the identified safe faults as CCA in Table 3. In short, the proposed technique identified 11.088% safe faults, which is two times more than when compared to the application-independent analysis.

More specifically, we have the following:

- The *regs* unit has configuration registers, whose value is written in the initialization phase. Since the UART configuration is fixed in CCA, some parts of the UART are unused; thus, several safe faults can be identified in this unit.
- Safe faults in the *transmitter* module originate from the configuration of the transmission format, such as the selected BAUD rate. Therefore, more safe faults can be found in this unit when the SW application is fixed, as in this work. Correspondingly, *transmitter fifo* is partially affected by these factors.
- Concerning the *receiver* module that is directly affected by the configuration registers, a significant amount of increase in the number of safe faults is observed. This mainly stems from the fact that the receiver module is responsible for generating interrupts. However, the CCA works in polling mode, meaning that no interrupt is used. Moreover, the receiver module has a modem configuration, which CCA does not need. By extension, *receiver fifo* is partly affected, similar to transmitter fifo.

Table 3. Safe faults in the UART IP.

UART Modules	Application-Independent		CCA	
	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults
Top	9	0.047%	19	0.099%
wb_interface	78	0.408%	78	0.408%
regs	357	1.867%	1003	5.246%
transmitter	67	0.350%	67	0.350%
uart_sync_flops	6	0.031%	6	0.031%
fifo_tx	101	0.528%	101	0.528%
receiver	171	0.894%	651	3.405%
fifo_rx	195	1.020%	195	1.020%
TOTAL	984	5.146%	2120	11.088%

6.2.3. Safe Faults in CAN

The same analysis is performed for the CAN module, which has 38,012 faults in total, and the results are provided in Table 4. In the application-independent analysis, the formal analysis tool can classify only 1.415% of all faults as safe. On the other hand, when the proposed approach is deployed, the amount of safe faults is increased to 12.909%, which is not negligible.

Similar to UART, the number of safe faults in CAN is directly affected by its configuration. In CCA, we configure the CAN to work in peliCAN mode, which has extended frame format messages. When the basiCAN mode is used, more safe faults can be identified. To put the results given in Table 4 more explicitly, we have the following:

- *Acceptance_code_mask* defines whether the corresponding incoming bit is compared to the respective bit in the *acceptance_code_regs*. Similarly, *bus_timing_regs* defines the values of the Baud rate prescaler and programs the period of the CAN system. Moreover, *clock_divider_regs* controls the clock frequency for the microcontroller and allows to deactivate the clock pin. In addition, the CCA works in polling mode, so

safe faults can be found in the *IRQ* registers. Consequently, all these registers should not be changed after the initial configuration; thus, this creates additional safe faults.

- *Bit timing logic* is directly affected by *bus_timing_regs* explained above, so the CCA originates some safe faults in this unit.
- *Bit stream processor* corresponds to the control and processing unit of the peripheral. It is a sequencer that controls the data stream between the transmit buffer, the receive fifo, and the CAN bus. Additionally, error-detection, arbitration, stuffing, and error-handling are done in this unit. In addition, the bit stream processor is affected by the configuration, such as working mode of the CAN, such as the listen-only mode or self-test mode. The CCA does not use these modes, which provide the safe faults shown in Table 4.
- *Acceptance filter* checks whether the message currently on the bus has to be stored by the peripheral or not. If the message is accepted, it is stored in the fifo. In other words, the bit acceptance filter and its fifo are related to *acceptance_code_regs* and *acceptance_code_mask*; therefore, the fixed content of these registers gives rise to safe faults.

Table 4. Safe faults in the CAN controller IP.

CAN Modules	Application-Independent		CCA	
	Safe Faults	Safe Faults w.r.t. Total Faults	Safe Faults	Safe Faults w.r.t. Total Faults
Top	10	0.026%	41	0.108%
can_registers	22	0.058%	769	2.023%
acceptance_code_regs	0	0.000%	52	0.137%
acceptance_mask_regs	0	0.000%	52	0.137%
bus_timing_regs	0	0.000%	26	0.068%
clock_divider_regs	11	0.029%	41	0.108%
command_reg	13	0.034%	57	0.150%
error_warning_reg	10	0.026%	74	0.195%
irq_en_reg	0	0.000%	15	0.039%
mode_regs	14	0.037%	53	0.139%
tx_data_regs	0	0.000%	115	0.303%
Bit Timing Logic	46	0.121%	299	0.787%
Bit Stream Processor	354	0.931%	2988	7.861%
can_crc_rx	0	0.000%	0	0.000%
Acceptance Filter	3	0.008%	256	0.673%
can_fifo	55	0.145%	69	0.182%
TOTAL	538	1.415%	4907	12.909%

6.2.4. Combined Results: Fault Simulation and Formal Analysis

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

As mentioned in Section 3.2, the fault simulation is not enough to classify all faults because workloads used for fault simulation cannot activate and propagate all faults. Therefore, some faults become undetected as a result of fault simulation. It is needed to analyze these undetected faults to check if the desired DC is reached. If the DC does not match the requirements, then the undetected faults must be re-analyzed using alternative methods, such as the proposed technique in this work. In short, the purpose of this step is to show that the proposed technique can increase DC to achieve the figures required by a given automotive safety integrity level.

In order to perform this analysis, we resorted to the software-based self-test (SBST) [5] approach in the form of STLs. In the considered scenario, the AutoSoC runs BareMetal-CCA in the field, and the STL, when activated, forces the processor to execute a proper

sequence of instructions. Then, a signature is produced based on the generated results, and the application can compare it with the expected results if there are faults.

The developed STL for the AutoSoC CPU is a combination of 57 test programs, partly taken from [8] and partly newly developed for this paper. Concerning the STL for CAN, we use the same test programs described in [28]. The STL was developed as a collection of tasks that can either operate independently or collectively, depending on the self-test time slot [37].

The following steps are applied:

- First, Str-Safe faults are identified using the Cadence® JasperGold® Functional Safety Verification (FSV) App.
- Second, we use the Cadence® Xcelium™ Fault Simulator to inject SA0 and SA1 faults at cell ports of the AutoSoC CPU and CAN modules, which run the STL as a workload. As a result, faults are classified as detected or undetected.
- Third, DC is calculated by using (1).
- Fourth, App-Safe faults are identified before being excluded from undetected faults. This process is incremental, always focusing on faults that were previously undetected.
- Finally, DC is calculated again with the newly achieved numbers using (1).

Figures 10 and 11 detail the results of the STL efficiency and uptrend in DC when App-Safe faults are identified. Concerning the analysis in CPU, Figure 10 shows that 8465 Str-Safe faults are identified in the beginning. Then, when fault simulation is deployed, 71,255 detected and 16,634 undetected faults are classified. After fault simulation, DC is 81.07%, calculated using (1). Then, by applying the proposed safe fault identification technique using formal methods, 5627 App-Safe faults are identified, i.e., undetected faults are reduced to 11,007. Using again (1), DC is increased to 86.62%. A similar analysis is performed in CAN as shown in Figure 11. As a result, DC is increased from 88.04% to 91.97%.

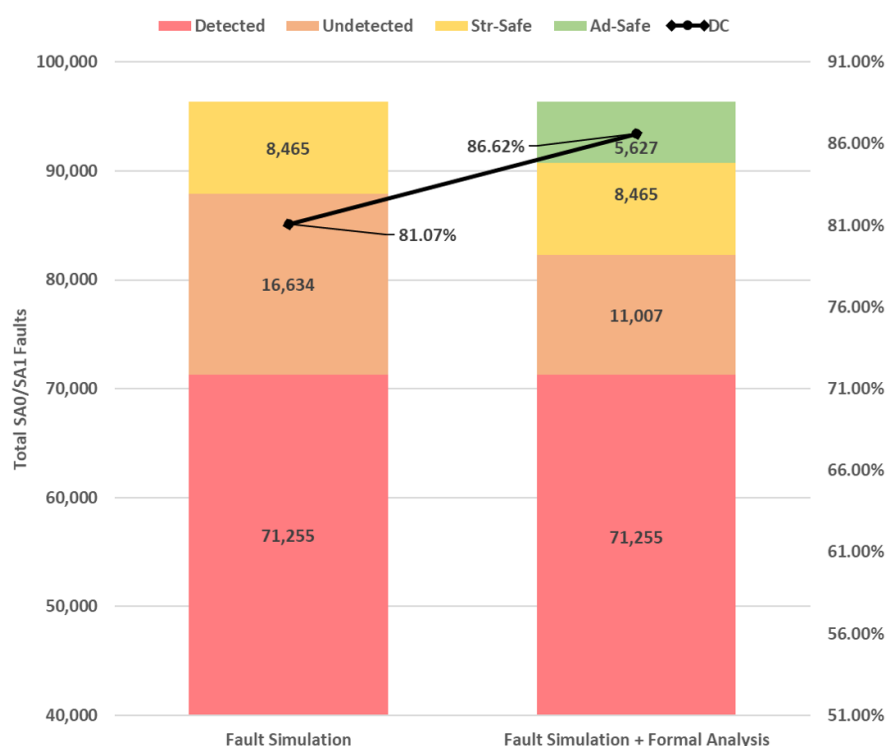


Figure 10. Combined results in CPU: uptrend in DC when fault simulation and formal analysis combined.



Figure 11. Combined results in CAN: uptrend in DC when fault simulation and formal analysis combined.

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

6.3. Discussion

Safety standards (e.g., ISO 26262) mandate the estimation of the achieved safety level, which in turn requires the identification of safe faults. This work provides a new technique for automatically identifying safe faults in the CPU and peripherals. The proposed technique can significantly reduce the cost and effort for safe fault identification, showing that the method can identify a significant number of safe faults. Safety standards (e.g., ISO 26262) mandate the estimation of the achieved safety level, which in turn requires the identification of safe faults. This work provides a new technique for automatically identifying safe faults in the CPU and peripherals. The proposed technique can significantly reduce the cost and effort for safe fault identification, showing that the method can identify a notable number of safe faults.

The main advantage of the proposed method is its automated approach for safe fault identification using the automotive representative hardware and software application. The method reduces the constraints of manual expert-based analysis, so the time and complexity of verification efforts are reduced simultaneously. This also helps reduce the time-to-market criteria, which is one of the biggest challenges of the IC design industry. Moreover, the proposed method is systematic and established based on logic simulation and formal analysis, supported by industrial-grade tools that make it suitable for the automotive industry and research on functional safety verification. It enables an accurate safety metrics evaluation; therefore, it allows compliance with ISO 26262 functional safety metrics. Concerning disadvantages, there is no analytical calculation regarding the number

of logic simulations to be run. We ran several logic simulations using different but realistic input data sequences in our work (as explained in Section 4.1.1) and stopped running new ones when the coverage reports were the same. Additionally, concerning the computational complexity of the proposed technique, it depends upon the number of faults that are being evaluated by the formal analysis.

7. Conclusions

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

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

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

In future work, we plan to analyze different automotive representative software applications with various scenarios to see the effect of the proposed method on the safety metrics. Furthermore, an FMEDA will be created, and safety metrics guided by ISO 26262 will be calculated when the proposed technique identifies safe faults. Additionally, different STLs that target the AutoSoC and peripherals will be deployed, and the increase in the safety level will be analyzed.

Author Contributions: Conceptualization, A.C.B. and F.A.d.S.; data curation, A.C.B.; formal analysis, A.C.B.; investigation, A.C.B. and F.A.d.S.; methodology, A.C.B. and F.A.d.S.; software, A.C.B.; writing—original draft preparation, A.C.B.; supervision, M.J., M.S.R., S.H. and C.S.; writing—review and editing, A.C.B., F.A.d.S., M.S.R., S.H., M.J. and C.S. All authors have read and agreed to the published version of the manuscript.

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

Data Availability Statement: The AutoSoC Benchmark Suite is open source and available at <https://www.autosoc.org> (accessed on 18 December 2021) Additionally, the original contributions presented in the study are included in the article, and further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ADAS	Advanced Driver Assistance System
ALU	Arithmetic Logic Unit
ASIL	Automotive Safety Integrity Level
ATPG	Automatic Test Pattern Generation
BDD	Binary Decision Diagram
BIST	Built-in Self-Test
CAN	Controller Area Network
CCA	Cruise Control Application
CPU	Central Processing Unit
COI	Cone-of-Influence
DC	Diagnostic Coverage
DUT	Design Under Test
ECU	Electronic Control Unit
FSV	Functional Safety Verification
IC	Integrated Circuit
IMC	Integrated Metrics Center
RTEMS	Real-Time Executive for Multiprocessor Systems
RTL	Register Transfer Level
SBST	Software-Based Self-Test
SoC	System-on-Chip
STL	Software Test Library
SW	Software
TCL	Tool Command Language
UART	Universal Asynchronous Receiver–Transmitter
XFS	Xcelium Fault Simulator

References

1. Jenihhin, M.; Sonza Reorda, M.; Balakrishnan, A.; Alexandrescu, D. Challenges of Reliability Assessment and Enhancement in Autonomous Systems. In Proceedings of the 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Noordwijk, The Netherlands, 2–4 October 2019; pp. 1–6. [\[CrossRef\]](#)
2. Munir, A. Safety Assessment and Design of Dependable Cybercars: For today and the future. *IEEE Consum. Electron. Mag.* **2017**, *6*, 69–77. [\[CrossRef\]](#)
3. Nardi, A.; Armato, A. Functional safety methodologies for automotive applications. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; pp. 970–975. [\[CrossRef\]](#)
4. International Standardization Organization. *ISO 26262 Road Vehicles—Functional Safety*, 2nd ed.; ISO: Geneva, Switzerland, 2018.
5. Psarakis, M.; Gizopoulos, D.; Sanchez, E.; Sonza Reorda, M. Microprocessor Software-Based Self-Testing. *IEEE Des. Test Comput.* **2010**, *27*, 4–19. [\[CrossRef\]](#)
6. Cantoro, R.; Firrincieli, A.; Piumatti, D.; Restifo, M.; Sanchez, E.; Sonza Reorda, M. About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications. In Proceedings of the 2018 IEEE 19th Latin-American Test Symposium (LATS), São Paulo, Brazil, 12–16 March 2018; pp. 1–6. [\[CrossRef\]](#)
7. Benso, A.; Di Carlo, S. The Art of Fault Injection. *Control Eng. Appl. Inform.* **2011**, *13*, 9–18.
8. da Silva, F.A.; Cagri Bagbaba, A.; Ruospo, A.; Mariani, R.; Kanawati, G.; Sanchez, E.; Sonza Reorda, M.; Jenihhin, M.; Hamdioui, S.; Sauer, C. Special Session: AutoSoC—A Suite of Open-Source Automotive SoC Benchmarks. In Proceedings of the 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 5–8 April 2020; pp. 1–9. [\[CrossRef\]](#)
9. SJA1000, Stand-Alone CAN Controller. Available online: <https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf> (accessed on 18 December 2021).
10. Alexandrescu, D.; Evans, A.; Glorieux, M.; Nofal, I. EDA support for functional safety—How static and dynamic failure analysis can improve productivity in the assessment of functional safety. In Proceedings of the 2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS), Thessaloniki, Greece, 3–5 July 2017; pp. 145–150. [\[CrossRef\]](#)
11. Lu, K.L.; Chen, Y.Y.; Huang, L.R. FMEDA-Based Fault Injection and Data Analysis in Compliance with ISO-26262. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Luxembourg, 25–28 June 2018; pp. 275–278. [\[CrossRef\]](#)
12. Juez, G.; Amparan, E.; Lattarulo, R.; Rastelli, J.P.; Ruiz, A.; Espinoza, H. Safety assessment of automated vehicle functions by simulation-based fault injection. In Proceedings of the 2017 IEEE International Conference on Vehicular Electronics and Safety (ICVES), Vienna, Austria, 27–28 June 2017; pp. 214–219. [\[CrossRef\]](#)

13. Fu, Y.; Terechko, A.; Bijlsma, T.; Cuijpers, P.J.L.; Redegeld, J.; Örs, A.O. A Retargetable Fault Injection Framework for Safety Validation of Autonomous Vehicles. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 69–76. [\[CrossRef\]](#)
14. Ferlini, F.; Seman, L.O.; Bezerra, E.A. Enabling ISO 26262 Compliance with Accelerated Diagnostic Coverage Assessment. *Electronics* **2020**, *9*, 732. [\[CrossRef\]](#)
15. da Silva, F.A.; Bagbaba, A.C.; Hamdioui, S.; Sauer, C. Flip Flop Weighting: A technique for estimation of safety metrics in Automotive Designs. In Proceedings of the 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 28–30 June 2021; pp. 1–7. [\[CrossRef\]](#)
16. Raik, J.; Fujiwara, H.; Ubar, R.; Krivenko, A. Untestable Fault Identification in Sequential Circuits Using Model-Checking. In Proceedings of the 2008 17th Asian Test Symposium, Sapporo, Japan, 24–27 November 2008; pp. 21–26. [\[CrossRef\]](#)
17. Syal, M.; Hsiao, M. New techniques for untestable fault identification in sequential circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2006**, *25*, 1117–1131. [\[CrossRef\]](#)
18. Liang, H.C.; Lee, C.L.; Chen, J. Identifying untestable faults in sequential circuits. *IEEE Des. Test Comput.* **1995**, *12*, 14–23. [\[CrossRef\]](#)
19. Condia, J.E.R.; Da Silva, F.A.; Hamdioui, S.; Sauer, C.; Reorda, M.S. Untestable faults identification in GPGPUs for safety-critical applications. In Proceedings of the 2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Genova, Italy, 27–29 November 2019; pp. 570–573. [\[CrossRef\]](#)
20. Augusto da Silva, F.; Bagbaba, A.C.; Hamdioui, S.; Sauer, C. Combining Fault Analysis Technologies for ISO26262 Functional Safety Verification. In Proceedings of the 2019 IEEE 28th Asian Test Symposium (ATS), Kolkata, India, 10–13 December 2019; pp. 129–1295. [\[CrossRef\]](#)
21. Bernardini, A.; Ecker, W.; Schlichtmann, U. Where formal verification can help in functional safety analysis. In Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Austin, TX, USA, 7–10 November 2016; pp. 1–8. [\[CrossRef\]](#)
22. Lai, W.C.; Krstic, A.; Cheng, K.T. Functionally testable path delay faults on a microprocessor. *IEEE Des. Test Comput.* **2000**, *17*, 6–14. [\[CrossRef\]](#)
23. Tille, D.; Drechsler, R. A Fast Untestability Proof for SAT-Based ATPG. In Proceedings of the 2009 DDECS '09—12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, Liberec, Czech Republic, 15–17 April 2009; p. 38–43. [\[CrossRef\]](#)
24. Long, D.E.; Iyer, M.A.; Abramovici, M. FILL and FUNI: Algorithms to Identify Illegal States and Sequentially Untestable Faults. *ACM Trans. Des. Autom. Electron. Syst.* **2000**, *5*, 631–657. [\[CrossRef\]](#)
25. Gursoy, C.; Jenihhin, M.; Oyeniran, A.S.; Piumatti, D.; Raik, J.; Sonza Reorda, M.; Ubar, R. New categories of Safe Faults in a processor-based Embedded System. In Proceedings of the 2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), Cluj-Napoca, Romania, 24–26 April 2019; pp. 1–4. [\[CrossRef\]](#)
26. Cantoro, R.; Carbonara, S.; Floridia, A.; Sanchez, E.; Sonza Reorda, M.; Mess, J.G. Improved Test Solutions for COTS-Based Systems in Space Applications. In *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*; Bombieri, N., Pravadelli, G., Fujita, M., Austin, T., Reis, R., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 187–206.
27. Narang, A.; Venu, B.; Khursheed, S.; Harrod, P. An Exploration of Microprocessor Self-Test Optimisation Based On Safe Faults. In Proceedings of the 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Athens, Greece, 6–8 October 2021; pp. 1–6. [\[CrossRef\]](#)
28. da Silva, F.A.; Bagbaba, A.C.; Sartoni, S.; Cantoro, R.; Sonza Reorda, M.; Hamdioui, S.; Sauer, C. Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs. In Proceedings of the 2020 IEEE European Test Symposium (ETS), Tallinn, Estonia, 25–29 May 2020; pp. 1–6. [\[CrossRef\]](#)
29. Maier, P.R.; Sharif, U.; Mueller-Gritschneider, D.; Schlichtmann, U. Efficient Fault Injection for Embedded Systems: As Fast as Possible but as Accurate as Necessary. In Proceedings of the 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, Spain, 2–4 July 2018; pp. 119–122. [\[CrossRef\]](#)
30. Clarke, E.; McMillan, K.; Campos, S.; Hartonas-Garmhausen, V. Symbolic model checking. In *Computer Aided Verification*; Alur, R., Henzinger, T.A., Eds.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 419–422.
31. Tasiran, S.; Keutzer, K. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test Comput.* **2001**, *18*, 36–45. [\[CrossRef\]](#)
32. Devarajegowda, K.; Servadei, L.; Han, Z.; Werner, M.; Ecker, W. Formal Verification Methodology in an Industrial Setup. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; pp. 610–614. [\[CrossRef\]](#)
33. Lach, J.; Bingham, S.; Elks, C.; Lenhart, T.; Nguyen, T.; Salaun, P. Accessible formal verification for safety-critical hardware design. In Proceedings of the RAMS '06—Annual Reliability and Maintainability Symposium, Newport Beach, CA, USA, 23–26 January 2006; pp. 29–32. [\[CrossRef\]](#)
34. OpenRISC 1000 Architecture Manual. Available online: <https://openrisc.io/or1k.html> (accessed on 18 December 2021).
35. RTEMS Real Time Operating Systems (RTOS). Available online: <https://www.rtems.org/> (accessed on 18 December 2021).

-
36. Chen, H.; Tian, J. Research on the Controller Area Network. In Proceedings of the 2009 International Conference on Networking and Digital Society, Guiyang, China, 30–31 May 2009; Volume 2, pp. 251–254. [[CrossRef](#)]
 37. Cantoro, R.; Sartoni, S.; Sonza Reorda, M. In-field Functional Test of CAN Bus Controllers. In Proceedings of the 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 5–8 April 2020; pp. 1–6. [[CrossRef](#)]