Determined-Safe Faults Identification

A step towards ISO26262 hardware compliant designs

Augusto da Silva, Felipe ; Bagbaba, Ahmet Cagri; Sartoni, Sandro; Cantoro, Riccardo; Reorda, Matteo Sonza; Hamdioui, Said; Sauer, Christian

DOI
10.1109/ETS48528.2020.9131568

Publication date
2020

Document Version
Final published version

Published in
2020 IEEE European Test Symposium (ETS)

Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs

Felipe Augusto da Silva*†, Ahmet Cagri Bagbaba*, Sandro Sartoni‡, Riccardo Cantoro‡, Matteo Sonza Reorda‡, Said Hamdioui† and Christian Sauer*

*Cadence Design Systems Munich, Germany
†Delft University of Technology Delft, The Netherlands
‡Politecnico di Torino Turin, Italy

***Abstract*—The development of Integrated Circuits for the Automotive sector imposes on major challenges. ISO26262 compliance, as part of this process, entails complex analysis for the evaluation of potential random hardware faults. This paper proposes a systematic approach to identify faults that do not disrupt safety-critical functionalities and consequently can be considered Safe. By deploying code coverage and Formal verification techniques, our methodology enables the classification of faults that are unclassified by other technologies, improving ISO26262 compliance. Our results, in combination with Fault Simulation, achieved a Diagnostic Coverage of 93% in a CAN Controller. These figures allow an initial assessment for an ASIL B configuration of the IP.**

Keywords - ISO26262; Safe Faults; Fault Injection; Formal Methods; Simulation; Functional Safety; Verification.

I. INTRODUCTION

The increasing complexity in automotive applications is causing a shift in the traditional design flow. An Integrated Circuit (IC) that implements safety-critical applications, such as autonomous driving, must incorporate mechanisms to reduce the risk of failures resulting in life-threatening situations. For such applications, the system must be able to detect an extremely high percentage of potential faults while already deployed in the field. In the most advanced automotive ICs, where millions of design components are susceptible to random hardware faults, this process becomes challenging. Also, the demands for fault detection during the operational life of the design requires the deployment of suitable test mechanisms, as Self Test Libraries (STL). In operational mode, Design for Testability (DfT) often is not an option, as it could disturb the intended functionalities. Today, the approach based on STLs is widely adopted in the automotive industry [1][2][3].

Usually, Fault Injection (FI) Simulations are deployed for evaluation of the fault effects in the operational mode. However, FI Simulation alone is not enough to fully classify all faults. For those which are not detected we must rely on alternative analysis methods that can prove whether they could disturb safety-critical functionalities or not (*Safe Faults*). Previous works [4] showed that the number of Safe Faults can be significant in real applications. In complicated designs, manual analysis of fault effects is an arduous task that requires extensive knowledge of the design functionalities. Therefore, there is a high demand for a systematic approach for the identification of Safe Faults, allowing the reduction of manual efforts and improving compliance with Functional Safety standards.

Fault Injection (FI) Simulation is a state-of-the-art method for Functional Safety Verification, being recommended by ISO26262. As such, several researchers explored the optimization of FI campaigns [5][6][7][8]. The main purpose is to show that fault effects are observable on safety-related outputs of the design. In case an injected fault is not observable, it must be re-analyzed. Nonetheless, observation or detection of all design faults is usually not possible. Therefore, alternate methods are necessary for the classification of residual faults. Formal Methods can be employed to leverage the classification of faults. The ability of formal techniques in analyzing the design behavior for all possible combinations of inputs can help to identify Safe Faults [9][10][11]. These faults cannot be tested by ANY valid test stimuli. Faults that are untestable can also be described as Structural-Safe Faults. The combination of FI Simulation and Formal techniques was also examined [12][13][14][15]. The mixed technologies approach is usually deployed to improve the classification of faults. However, even with the identification of Detected and Structural-Safe Faults, there are still residual faults that require further classification. To avoid manual analysis of fault effects and still fulfill ISO26262 requirements, a different methodology is needed.

Our work tackles the classification of residual faults. We propose a methodology that identifies design elements where a fault cannot disturb the safety-critical outputs of the design. In case the effect of a fault does not affect safety-related functionalities, there is no chance of Safety Goal violations. Therefore, these faults can be classified as Determined-Safe. Different from Structural-Safe Faults which cannot be tested by any functional test stimuli, Determined-Safe Faults may affect the output of the design. However, they cannot affect safety-critical functionalities. Initially, we deploy code coverage techniques to identify design elements that are not exercised during functional verification. The candidates are examined by code inspection and simulation. If confirmed that the candidates are not safety-related, they are translated into formal rules. Finally, we configure all the rules in a Formal analysis tool for the identification of Determined-Safe Faults. The main contributions of this work are:

- A systematic approach for classification of Faults that cannot affect safety-critical functionalities;

978-1-7281-4312-5/20/$31.00 ©2020 IEEE

- Demonstration of the proposed methodology using an automotive CAN Controller IP;
- Improving the fault classification to 93% of Diagnostic Coverage, achieving ASIL B requirements out of the box.

The rest of the paper is organized as follows: Formal techniques for identification of Safe Faults are introduced in Section II. Section III describes the proposed methodology. Section IV explains the validation process and discusses our results. Section V concludes.

II. FAULT CLASSIFICATION BY FORMAL TOOLS

Fault classification is a strenuous task. A fault can only be labeled as Safe if one can prove that it cannot be tested by ANY functional test stimuli. The formal analysis appears as a good alternative for this purpose since it is not limited to a specific time or state. Instead, the scope is global, and every evaluation context and test stimuli is considered [9]. Consequently, formal analysis can exhaustively prove that a fault can never produce any failure. This class of untestable faults can be classified as Structural-Safe.

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

A. Structural Analysis

The Structural Analysis aims to determine the testability of faults. The testability of the faults is determined by verifying the physical characteristics of the design. Figure 1 illustrates the examination applied by the Structural Analysis.

Figure 1 represents a circuit with combinational logic (g), inputs (in), outputs (out) and fault targets (f). Considering this circuit, it is possible to define the following fault behaviors by applying Structural Analysis:

1) As the only Observation Point (strobe) configured for the fault analysis is 'out0', any fault that is outside of its Cone of Influence is considered Untestable. For that reason, any fault in 'f1' is Structural-Safe as there is no physical connection between the fault location and the strobe.
2) Depending on the characteristics of 'g1' drivers, it is possible to define the activatability of 'f2'. For example, if 'g1' always output the logic value one, 'f2' would not be activatable for Stuck-at-1 faults. Consequently, a Stuck-at-1 fault in 'f2' would be Structural-Safe.
3) Characteristics of the combinational logic 'g2' could block propagation of a fault in 'f3'. If, for example, 'g2' is an AND gate, with one of the inputs always set with the logic value zero, the effect of a fault in 'f3' would never propagate to 'out0'. Therefore, 'f3' would be Structural-Safe for Stuck-at-1 and Stuck-at-0 faults.

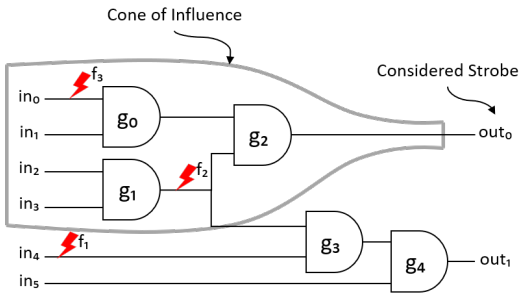

Fig. 1. Structural Analysis Example.

B. Formal Analysis

The Formal Analysis deploys formal techniques to investigate the behavior of a design under fault. The fundamental theory consists in creating a representation of the boolean function implemented by the design under test, where formal proves can be deployed. Modern Formal tools employ different formal techniques to achieve better performance. Although details of implementation are not disclosed, common forms of design representation are Binary Decision Diagrams (BDDs) [16] and Multiway Decision Graphs (MDGs) [17].

Two copies of the design model are built for formal analysis: the Good Machine and the Bad Machine. The same inputs and constraints are deployed on both models. Fault effects are applied in the Bad Machine only and the Strobe point of both copies are monitored. A difference in the Strobe Points indicate the propagation of the fault.

The Formal Analysis deploys formal methods to determine the Activation and Propagation of faults. Activation Analysis indicates whether the fault can be functionally activated by any combination of inputs. Propagation Analysis verifies if there is a combination of inputs that provoke fault propagation. Formal Analysis will classify the faults, which were not previously classified by the Structural Analysis, in three groups:

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

Formal properties to perform the analysis are automatically generated and verified with respect to all possible input stimuli. The Formal Analysis relies on formal properties and verification to prove the properties to be true.

Formal verification techniques are resource hungry and limited due to the state explosion problem. For that reason, the analysis of formal properties cannot find results for all fault targets. Therefore, the residual faults still require an alternative classification methodology.

III. DETERMINED-SAFE FAULTS

The ISO26262 Hardware Architectural Metrics determines the effectiveness of designs to cope with random hardware failures [18]. The failures addressed by these metrics are limited to elements that can contribute to the violation of safety

goals. Safety goals define the required mitigation of hazardous events to avoid unreasonable risks caused by malfunctions. During the system development phase, safety goals will be decomposed into a Functional Safety Concept that defines the requirements for the hardware architecture. However, the development of a hardware design demands additional components that are not related to the safety concept. These components will decrease the compliance to Hardware Architectural Metrics, even though in case of faults, they may not violate safety goals. For that reason, these components can be identified by their potential to disrupt safety goals and, if applicable, determined safe.

Determined-Safe Faults cannot disturb safety goals. Different from Structural-Safe Faults that cannot be tested by ANY functional test stimuli, Determined-Safe Faults may affect the output of the design. However, they cannot affect safety-critical functionalities. Common Determined-Safe fault targets are design parts not used in operational mode. The identification of these faults usually requires the judgment of hardware design experts.

A. Determined-Safe Candidates

In this section, we define a methodology to support the identification of Determined-Safe Faults. Our methodology deploys code coverage techniques to identify design elements that are not fully used during the design simulation. Code coverage is a method of assessing to what extent test cases exercise the design. Since this analysis relies on the simulation results, it is critical to employ representative test cases. In general, Functional Safety Verification is performed at later stages of the design life-cycle, after functional verification is completed. Therefore, we can assume that the design is available in RT and Gate level, and also comprehensive test cases are available for the identification of Determined-Safe Faults.

The initial step is to simulate the design under test, with all the available test cases, collect code coverage data, and generate the coverage reports. Our methodology does not depend on a specific tool. However, the selected tool-set should include code coverage analysis. Next, we analyze the reports to check the results for block and toggle coverage. Block coverage determines whether test scenarios exercise the statements in a block. A block is a series of sequential statements without delays or control flow statements (if, case, wait, while, among others). In other words, a block is a specific state in a state machine. Toggle coverage measures the activity of the signals in the design during the simulation. It provides information on untoggled signals or signals that remain constant during the simulation.

The metrics from the code coverage provide candidates for Determined-Safe Faults. For instance, by recognizing states that are never activated, as a result of block coverage, we can identify design modes that are not related to safety functionalities. Similarly, signals that are untoggled can highlight important details of the design, like invalid configurations, not utilized functions, status monitors, among others. The

a)

```
always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_irq <= 1'b0;
  else if (error_status ^error_status_q)
    error_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    error_irq <=#Tp 1'b0;
end
```

b)

Name		
read_irq_reg	0	0
error_irq	1	0

Fig. 2. a) Block Coverage example - b) Toggle Coverage example.

combination of toggle and block coverage usually provides further information about specific functionalities. For example, the missing toggle in a control signal may be responsible for never activating a block in a state machine. Also, by bypassing a specific state, another signal may not be toggled. Figure 2 illustrates an example of the correlation between the toggle and the block coverage. The block coverage (Figure 2-a) shows a block that was never activated. Since the last ”else if” statement is always false, the ’error_irq’ is not set to zero. In Figure 2-b, the result of toggle coverage shows that the control signal ’read_irq_reg’ never toggles, validating the block coverage. Additionally, the coverage confirms that the signal ’error_irq’ has one rising toggle but never toggles back to zero.

In this example, the coverage results trigger an investigation, where we can determine that the interrupt requests (IRQ) error register is never read by the application. Next, we need to verify if this behavior is expected, and then we can decide if a fault that affects the value of the IRQ error register can be considered safe. Each candidate identified during the code coverage requires an investigation over simulation and source code. The coverage result by itself is not enough to identify the potential Determined-Safe Faults. Nonetheless, it indicates candidates that can facilitate the manual classification of such faults. After the determination of the candidates, we need to translate their behavior into a set of formal rules, allowing identification of the actual Determined-Safe Faults by Formal methods.

B. Formal Identification of Determined-Safe Faults

The identification of Determined-Safe Faults will deploy the same techniques described in Section II for the identification of Structural-Safe Faults. The difference is that the formal environment will incorporate the formal rules retrieved from the code coverage analysis. By constraining the environment, we enable the tool to evaluate the design in a well-specified configuration, increasing the potential for identification of Safe Faults. Additional Safe Faults will be classified as Determined-Safe, as they are Safe considering the functional constraints included in the environment.

The design elements identified during the code coverage must be translated into *assume statements* or *fault-propagation barriers*. *Assume statements* enable constraints configuration for formal analysis. When an expression is assumed, the formal verification tool constrains the design inputs accordingly. The role of the assume construct is useful in the confirmation of the design functional configuration. Also, by configuring the expected behavior of the design, we increase the capacity of Safe Faults identification by limiting the test stimuli space. *Fault-propagation barriers* are design elements that can block the propagation of a fault. Faults that propagate only to certain elements may not affect safety-critical functionalities. Consequently, these faults can be Determined-Safe. For example, a counter that monitors the number of transmissions is not read by the transmission controller. In that case, a failure in the monitor does not alter the design functionality. For that reason, this counter can be configured as a fault-propagation barrier, and all faults that can only affect its value can be Determined-Safe.

In most cases, the Determined-Safe Candidates translation into formal environment constraints will consider the element type. Input ports of the design instances are suitable candidates to *assume statements*. Output ports, on the other hand, are better candidates for *fault-propagation barriers*. Internal signals like 'regs' and 'wires' need further analysis of the Gate-Level representation of the hardware, as they may be modified by synthesis. Nevertheless, for each environment constraint, we must confirm the assumptions by analysis of the RTL code, simulation of the design, and understanding of the expected design functionalities. An over-constrained formal environment would cause false-positives, invalidating the results.

After confirmation of the environmental constraints, we generate a file for the set-up of the Formal Analysis Tool. The set-up file must include all *assume statements* and *fault-propagation barriers*. With the set-up file in place, we repeat the formal analysis to identify the Determined-Safe Faults.

IV. RESULTS

A. Test Case

To validate the proposed methodology, we targeted a design that is representative of the challenges of the automotive industry. For that reason, the adopted peripheral is an open hardware implementation of the SJA1000 CAN Controller, developed by Philips in the early 2000s. The selected controller implements the *BasiCAN* and the *PeliCAN* Modes. The *BasiCAN* Mode supports communication in Normal Mode with a second CAN node. The *PeliCAN* Mode supports CAN 2.0B protocol, which includes functionalities as Self-Test and Listen-Only Modes.

The test of the CAN Controller considered for this work employs a Software-Based Self-Test (SBST) approach, leading to the creation of a Software Test Library (STL). To enable the execution of the STL and emulate a realistic configuration, the CAN Controller is integrated into an OpenRISC OR1200 SoC. By deploying a full SoC, we can store the test program in a memory and control the execution of the STL during idle intervals. The complete test environment comprises two OR1200 SoCs. Each SoC is configured with a different test program and connected through a simplified version of the CAN bus avoiding the implementation of the transceiver. Instead, the resulting bus consists of the two Tx signals connected into an AND gate whose output is then connected to each Rx pin. The environment can be configured with RT or Gate level representations of the CAN Controller.

TABLE I
FAULT INJECTION RESULTS.

Fault Target	SA(1/0) Faults	Undetected Faults	Detected Faults	Diagnostic Coverage
CAN Controller	38,012	5,005	33,007	86.83%

The STL was developed as a collection of tasks that can either operate independently or collectively, depending on the self-test time slot [19]. The following tasks are available as part of the STL:

- *Bitrate Test*: aims to test the timing related modules by employing different bitrates;
- *Normal Mode Test*: tests the BasiCAN and PeliCAN Normal Modes by transmitting and receiving messages with a fixed bitrate;
- *Self-Test Mode* and *Listen-Only Mode Tests*: while one node is in Self-Test mode the other one must be in Listen-Only Mode and vice versa;
- *FIFO Test*: tests the FIFO module by filling it and emptying it while receiving several messages;
- *Errors Test*: tests error conditions due to bitrate mismatches;
- *Arbitration Loss Test*: tests arbitration loss conditions, achieved when one node stops transmitting a message due to a higher priority message being transmitted on the bus;
- *Acceptance Filter Test*: tests the acceptance filter logic that decides whether a message has to be stored in the internal memory or not.

To validate the ability of the design to cope with random hardware faults, a Fault Injection campaign was executed. We used the Cadence® Xcelium™ Fault Simulator (XFS) to manage the fault campaign execution. The XFS was configured to inject SA0 and SA1 faults at every cell port of the Gate-Level representation of the CAN Controller. Table I shows the Fault Injection results. Even though the deployed STL achieves a good fault coverage (86.83%), there are still over 5,000 undetected faults. These faults must be classified to allow compliance with the requirements of ISO26262.

B. Classification of Determined-Safe Faults

During the analysis of the CAN Controller, the candidates for Determined-Safe Faults revealed some similarities. According to the intended functions, we could classify the candidates. First, several signals were constant during the simulation of the design. From those, nine are responsible for the configuration of the CAN Controller to operational

TABLE II
FORMAL ANALYSIS RESULTS.

Formal Analysis	SA(1/0) Faults	Structural Safe	Determined Safe	Total Safe
CAN Controller	38,012	539	1,996	2,535

mode. These signals were translated into *assume statements* in the constraints environment file. The combination of toggle and block coverage also revealed not used functionalities. The simulated workload does not enable modes like single-shot transmission, overload requests, and early transmission. Each of these cases must be individually analyzed. We need to define, based on the development requirements and safety goals, if these functionalities should be available in operational mode. As previously stated, our initial assumption is that the functional verification environment is available. Therefore, we can conclude that these modes are not intended in the current version of the CAN Controller. This assumption is reflected in the constraints environment file by the *assume statements* and *fault-propagation barriers*. Finally, the CAN Controller contains registers that monitor several statuses. Some of those are never read by the CPU. A misleading value in a monitor or counter that is never read by the application may not affect the expected functionality. Once again, safety goals should be verified to confirm that the CPU is not supposed to monitor these statuses. We have selected five status registers to be translated as *fault-propagation barriers* in the constraints environment file.

The constraints environment for the identification of Determined-Safe Faults on the CAN controller consisted of 10 *assume statements* and 18 *fault-propagation barriers*. We have examined the function of each included item by RTL code investigation and monitoring of the signals during the simulation. Also, some of the RTL internal signals needed to be traced to wires in the Gate level representation of the hardware to be included in the constraints environment.

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

C. Combined Results

The results of the Fault Injection and Formal Analysis can be combined to improve the Diagnostic Coverage. Faults that cannot disturb safety-critical functionalities, Structural and Determined-Safe, can be removed from the fault list. Each possible fault target in a design must be analyzed and classified. The annotation of the faults usually starts with Formal Analysis to identify Structural-Safe Faults. The remaining faults are simulated and, when applicable, annotated as Detected. If the desired Diagnostic Coverage is achieved, the process ends. Otherwise, the residual Undetected faults must be re-analyzed. The Determined-Safe classification is an alternative to annotate the remaining Undetected faults and increase the overall Diagnostic Coverage of the design. The Diagnostic Coverage is calculated by the formula:

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

where DC is the Diagnostic Coverage, Detected are faults annotated as detected by FI Simulation, Total is the number of faults, and Safe represents the Structural and Determined-Safe Faults annotated by the Formal tool.

Figure 3 details the results of the various analysis steps. The graph illustrates the faults classification contribution achieved during Fault Injection, Structural-Safe, and Determined-Safe analysis. The process is incremental, always focusing on faults that were not previously classified. Also, Figure 3 displays the calculated Diagnostic Coverage at each step. Finally, the last column illustrates the results when all fault analyses are combined. As previously explained, we apply Formal methods to decrease the number of not classified faults. As additional Safe Faults decrease the denominator in (1), the results from the Formal analysis cause an increase in the Diagnostic Coverage.

Even with the increased fault classification, there are still Undetected faults that require further analysis. The classification of the residual faults could be achieved by improving the STL coverage, or by creating additional formal rules to increase the number of Determined-Safe Faults. The next step of our work is to propose automation techniques that can facilitate the analysis and improve even further the fault classification.

The proposed methodology appears as a promising alternative for the classification of residual faults. We define a systematic approach that allows the identification of Safe Faults based on two well-established techniques. The identification of these faults usually relies on reliability experts and requires deep knowledge over the system functionalities. This manual analysis process is strenuous and prone to errors. Our methodology is a step towards the automation of the identification of Safe Faults. By deploying the proposed methodology, we were able to classify 2,535 additional faults, resulting in a DC improvement of around 6%. With a final DC of 93.04%, the CAN Controller achieves the requirements for an automotive ASIL B hardware component as-is, i.e., without design modifications.

V. CONCLUSIONS

Functional Safety Verification is one of the most challenging steps for Integrated Circuit (IC) compliance with ISO26262. The severe demands for tolerance to random faults are a hurdle

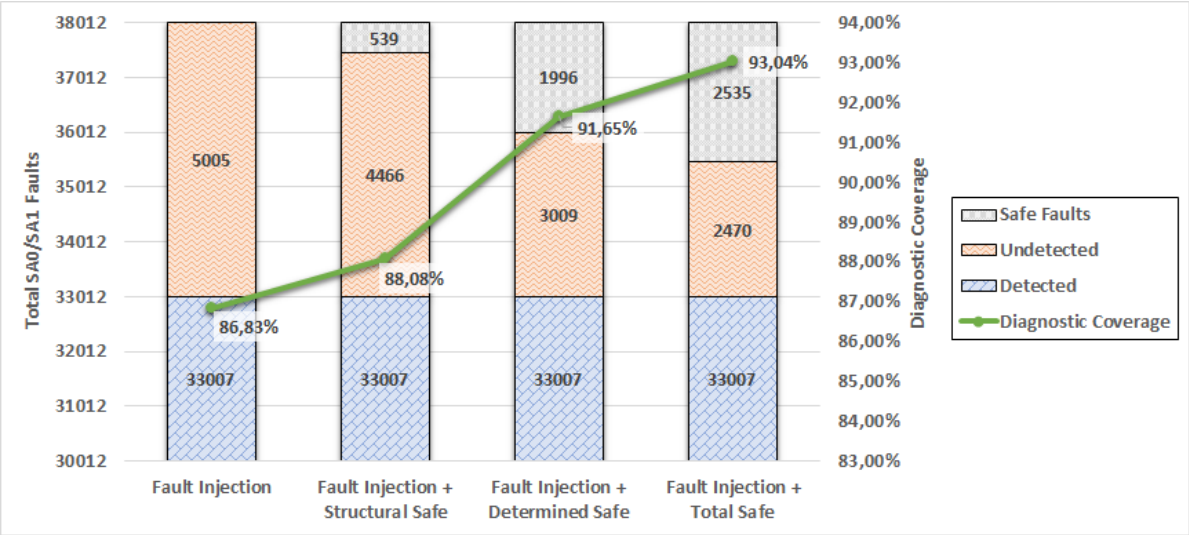

Fig. 3. Combined Results

for ICs targeting safety-critical applications. Fault analysis, as part of this process, becomes an extensive procedure that is usually repeated numerous times and requires manual inputs from specialists to achieve safety metrics. We propose a methodology that deploys code coverage and Formal analysis, as a step towards automation in Safe Faults identification. First, we identify design elements where a fault cannot disturb safety-critical functionalities. Next, those elements are translated into formal rules that are configured in a Formal analysis tool for the identification of Determined-Safe Faults. The additional classification of residual faults is necessary for compliance with ISO26262. Our methodology, in combination with Fault Simulation, was applied to a CAN Controller IP, resulting in a Diagnostic Coverage of 93%. The proposed methodology appears as a promising alternative for residual faults classification without relying solely on manual analysis.

ACKNOWLEDGMENT

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

REFERENCES

[1] ARM, "Development tools and software - Software Test Libraries," https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries, 2019.

[2] Cypress, *AN204377 FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library*, 2017.

[3] Microchip, *DS52076A 16-bit CPU Self-Test Library User's Guide*, 2012.

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

[5] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, nov 2017.

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

[7] D. Alexandrescu, A. Evans, M. Glorieux, and I. Nofal, "EDA support for functional safety — How static and dynamic failure analysis can improve productivity in the assessment of functional safety," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, jul 2017.

[8] Y.-C. Chang, L.-R. Huang, H.-C. Liu, C.-J. Yang, and C.-T. Chiu, "Assessing automotive functional safety microprocessor with ISO 26262 hardware requirements," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. IEEE, 2014.

[9] J. Raik, H. Fujiwara, R. Ubar, and A. Krivenko, "Untestable fault identification in sequential circuits using model-checking," in *2008 17th Asian Test Symposium*. IEEE, nov 2008.

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

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

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

[13] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Efficient methodology for ISO26262 functional safety verification," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, jul 2019.

[14] S. Marchese and J. Grosse, "Formal fault propagation analysis that scales to modern automotive socs," in *2017 Design and Verification Conference and Exhibition (DVCon) Europe*, 2017.

[15] A. Bernardini, W. Ecker, and U. Schlichtmann, "Where formal verification can help in functional safety analysis," in *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD*. ACM Press, 2016.

[16] G. Cabodi and M. Murciano, "BDD-based hardware verification," in *Formal Methods for Hardware Verification*. Springer Berlin Heidelberg, 2006, pp. 78–107.

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

[18] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 5: Product development at the hardware level*, International Standardization Organization Std., Dec. 2018.

[19] R. Cantoro, S. Sartoni, and M. Sonza Reorda, "In-field functional test of CAN bus controllers," in *2020 IEEE VLSI Test Symposium (VTS) - (to appear)*. IEEE, 2020.