Complex Safety Mechanisms Require Interoperability and Automation For Validation And Metric Closure

Daeseo Cha, Principal Engineer, Samsung Korea, (dscha@samsung.com)

Vedant Garg, Product Architect, Siemens EDA, USA (vedant.garg@siemens.com)

Ann Keffer, Product Manager, Siemens EDA, USA (ann.keffer@siemens.com)

Arun Gogineni, Software Architect, Siemens EDA, USA (arun.gogineni@siemens.com)

James Kim, Staff Application Engineer, Siemens EDA, South Korea (james.kim@Siemens.com)

Woojoo Space Kim, Principal Engineer, Samsung Korea, (space.kim@samsung.com)

Kunhyuk Kang, Corporate VP, Samsung Korea, (kunhyuk.kang@samsung.com)

Seonil Brian Choi, Corporate VP, Samsung Korea, (seonilb.choi@samsung.com)

Abstract: *The race to autonomous mobility among the automobile manufacturers is driving the evolution of the underlying semiconductors. As a result, semiconductor technologies are moving towards higher densities and lower operating voltages, and this migration is introducing increasing sensitivity to random hardware failures — the failures which occur unpredictably over a semiconductor's lifetime. Modern cars deploying ADAS and AV features rely on these digital and analog systems to perform critical real-time applications. This reliance has led to a concern over validation of these systems, and the question: are they* safe*?*

One practice to achieve or maintain a safe state is running an exhaustive fault campaign to test the effectiveness of a safety architecture's ability to detect faults or control failures. While traditional approaches may have been satisfactory in the past, the increased size and complexity of automotive designs with the large number of faults that need to be tested make performing safety verification using a single technology impractical.

Developing an optimized safety methodology with specific fault lists automatically targeted for simulation, emulation and formal is challenging. Another challenge is consolidating fault resolution results from various fault injection runs for final metric computation. Figure 1 shows some of the optimization techniques in a safety flow.

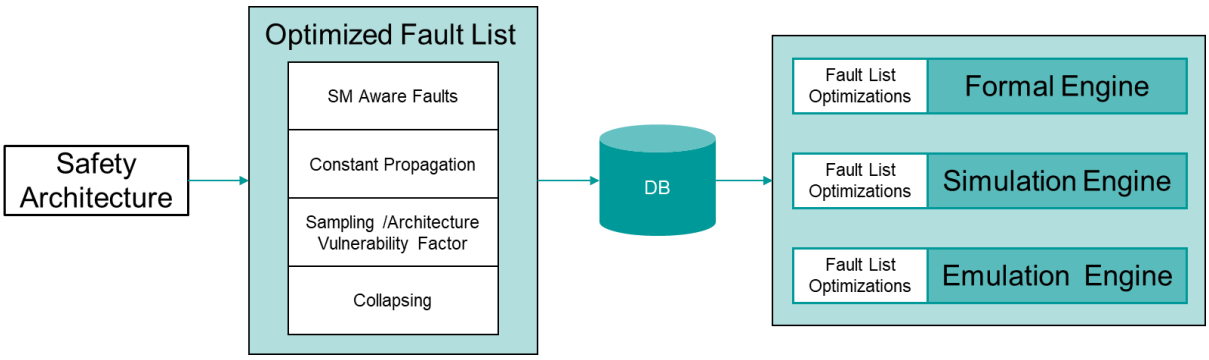

Figure 1: Fault List Optimization Techniques

In this paper we will discuss the details of the functional safety methodology we used for this application using an SoC level automotive test case, and we will show how our methodology produces a scalable, efficient safety workflow using optimization techniques for fault injection using formal, simulation, and emulation verification engines.

I. Introduction

There are several approaches to achieving or maintaining a safe state for safety architectures. A traditional approach to achieve a safe design is by running an exhaustive fault campaign to test the effectiveness of a safety architecture's ability to detect faults or control failures. While an exhaustive approach may have been satisfactory in the past, the increased size and complexity of automotive designs, along with the large number of faults that need to be analyzed, make it impractical to perform exhaustive, safety verification using a single technology. We will address this challenge by describing optimizations used today in the analysis and validation phases of the safety workflow for safety architectures.

II. Safety Analysis

Achieving safety for semiconductors and systems is defined by the ISO 26262 standard, which addresses the needs of electrical and electronic systems within road vehicles. Part 5 of the standard, which focuses on product development at the hardware level, states how hardware architectures need to be evaluated against the requirements for fault handling by ensuring the probabilities of random hardware failures are rigorously analyzed and quantified via a set of objective metrics [1].

The standard uses the automotive safety integrity level (ASIL) as the risk classification system to define the level of safety for road vehicles. ASIL is used to measure safety for systems and semiconductors, and ASIL metrics are assigned to components and subcomponents that are safety related in a semiconductor design. ASIL uses several metrics to analyze, quantify, and qualify the safeness of a design, such as failures in time (FIT), diagnostic coverage (DC), single point failure metric (SPFM), latent fault metric (LFM), and probabilistic metric for random hardware failures (PMHF). See Table 1 for metric definitions.

Metric	Acronym	Definition
Probabilistic Metric for Random Hardware Failures	PMHF	PMHF is the determination of the target value which describes a measurement for the probability regarding the random hardware failure (summary of all possible failures (single point and latent))
Failures in Time	FIT	The FIT rate of a device is the number of failures that can be expected in one billion (10^9) device-hours of operation
Diagnostic Coverage	DC	DC is the measurement of the effectiveness of the diagnostics or safety mechanisms in the system. It expresses the systematic capability of the safety mechanisms and is measured by dividing the failures detected/controlled by the safety mechanisms by the total failures in the system
Single Point Fault Metric	SPFM	The SPFM reflects if the coverage provided by safety mechanisms for single-point faults and residual faults in the hardware architecture is sufficient
Latent Fault Metric	LFM	The LFM reveals whether the coverage by the safety mechanisms for latent faults in the hardware architecture is sufficient

Table 1: Definition of ASIL metrics

Determining the level of a designs safety comes down to validating the effectiveness of the safety mechanisms in the safety critical system by determining the coverage achieved for the SPFM. ASIL determines the highest level of coverage is an SPFM of greater than or equal to 99% defined as ASIL D. See Table 2 for ASIL targets for SPFM.

	ASIL B	ASIL C	ASIL D
SPFM	>= 90%	>= 97%	>= 99%

Table 2: ASIL derivation of the target SPFM value

III. Optimizing the Safety Workflow

Optimizing the safety workflow is challenging. One approach to optimization is using an EDA analysis tool during the analysis phase to validate expert judgement and compute ASIL metrics through structural analysis that provide accurate metrics. By using an analysis tool on RTL blocks as soon as they are available, verification and safety engineers can get an indication of issues in the safety architecture early in the safety workflow. Getting as much safety testing done early at the RTL reduces the time spent in fault simulation (which is resource intensive, time consuming, and relies on good stimulus) and optimizes the entire safety workflow by lowering overall project cost and accelerating time-to-certification.

An exhaustive fault campaign — or determining the percentage of faults that will turn into failures for all nodes in the design — is a daunting task. This means injecting tens of thousands to millions of faults into hundreds of tests, simulating them to their outputs, and comparing the results for thousands to millions of nodes in the design. Fault list optimization techniques can help address this challenge. Fault list optimization can include adoption of techniques such as safety mechanism aware faults, fault collapsing, identifying faults that won't propagate, statistical random sampling, and architecture vulnerability factors. Formal techniques are a powerful method for determining the testability of faults by verifying, for example:

- If there is a physical connection between the fault location and the observation points
- If the signals that drive the fault node allow activation of the fault
- If the fault could be observable in at least one strobe of the design [2]

Fault lists can be further optimized for the specific engine used for fault injection. Fault injection engines used in the safety workflow can include formal, simulation, and emulation. Each engine has specific benefits and can reduce overall fault injection time when used together in a safety workflow. Table 3 is a high-level guide to the benefits of formal, simulation, and emulation for fault injection.

Formal	• Exhaustive, smart • No test bench required • Optimized for fault classification of safety mechanisms
Simulation	• Concurrent, parallel fault injection • Best for blocks and smaller IP • Optimized for hardware and software test libraries
Emulation	• Fast • Large SoCs, software driven designs • Optimized for end-to-end hardware safety mechanisms

Table 3: Benefits of formal, simulation and emulation for fault injection

IV. Automated Fault Injection Flow

ISO 26262, Chapter 11 explains that fault classification at the semiconductor component level can be achieved by combining formal, simulation, and emulation techniques. One of the biggest challenges is to make this methodology efficient and automated with respect to fault generation, choosing the classification technique (fault injection and formal), and combining the final metric results.

Figure 2 shows our execution methodology as a three-step flow for overall efficiency, resulting in reduced time to final metrics.

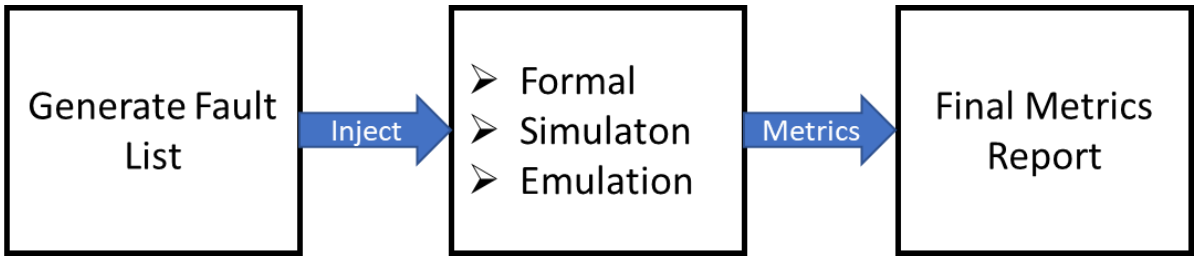

Figure 2: Three step workflow for efficient fault injection

Step 1: Generate the optimized fault list

Using fault optimization methods, such as fault collapsing for redundant faults, helps create an optimum fault list. Fault list optimization analysis should be done on each node with respect to all the safety mechanisms protecting the design. A failure modes and effects analysis (FMEA) can help create an optimized fault list for each failure mode (FM) of the design while excluding all the faults that are non-safety critical and do not violate the safety goal. This approach of bucketing fault lists for FMs helps automate the next step, which is fault injection.

Step 2: Fault injection and classification

The goal of fault injection is to exercise the random injection of faults and validate the effectiveness of the safety mechanisms. After faults are injected, they may propagate and can violate the safety goal. If the safety goal is violated, then the system should verify whether this violation was detected by a safety mechanism or not. ISO 26262 defines the fault classifications following fault injection as follows: single point fault (SPF), residual fault (RF), multi point fault latent (MPFL), multi point fault detected (MPFD), and safe fault (SF).

Faults which are safety related but do not have any protection from safety mechanisms are directly deemed as SPFs. For the remaining faults and their categorization, we will be using common formal, simulation, and emulation techniques.

Using Formal Analysis

Formal technology can be used before and after fault simulation to quickly identify and eliminate non-propagatable faults with the intent of reducing simulation and debug time. Formal tools use proof problems to assess if one or more signals in the corrupted design — typically referred to as observation points or safety-critical signals — have a different functional behavior compared to the original fault-free design. Formal analysis is equivalent to examining all input sequences to provide rigorous evidence that a certain fault is safe. Fault detection analysis (FDA) requires as inputs a list of observation points, a fault list, and a an RTL or gate-level design. It automatically selects the appropriate proof methods and strategies to find most of the safe faults, while aborting the analysis of hard-to-prove faults, thus minimizing effort spent on inconclusive results and debugging. Runtimes for FDA range from minutes to several hours, with overnight runs being acceptable for complex designs.

Using Fault Simulation

Analysis of fault injection by simulation is widely used and is a traditional method of fault classification. The KaleidoScope fault simulator analyzes RTL or gate level designs based on the given test inputs and injects faults to simulate a fault's behavior. The effect that a fault causes in the design is determined by comparing the behavior of the design with and without faults. It creates a "good simulation" that captures a fault-free behavior of the design. The values from the good simulation are stored for reference.

For fault injection with alarms and observation points as inputs, faults are injected with stimulus and the fault simulator records both if the fault fires the alarm and if the fault is observed. For each fault in the fault list, where the faulty behavior is simulated, the observation points are compared against the reference values generated during the good simulation. The fault simulator classifies and updates the results in the fault database with attributes.

Using Emulation

Fault emulation with the Veloce Fault App is required under conditions where safety mechanism complexity makes runtimes too lengthy for fault simulation. Software safety mechanisms that require full chip SoC and full software stacks to effectively achieve fault campaign closure are key targets for emulation. During synthesis, the emulator maintains a 1-to-1 gate mapping of the RTL and netlists with simulation. This step allows for net and memory element name reference consistency between the various backend execution engines. Stimulus vectors are also mapped and executed in the same 1-to-1 fashion, providing efficient cross engine debug.

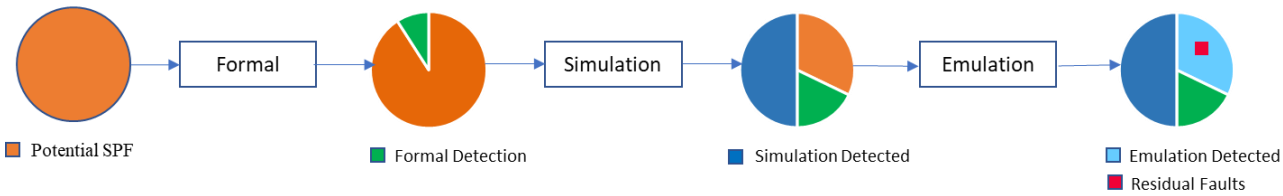

Figure 2: Fault resolution update after fault injection

Step 3: Generating the metrics report

The final step of this methodology is to consolidate the results from formal, fault simulation, and fault emulation to produce the overall safety metrics. The fault classification and the bucketing of faults are used for calculating the ASIL metrics — such SPFM, LFM. and PMHF — needed for the failure modes and effects diagnostics analysis (FMEDA).

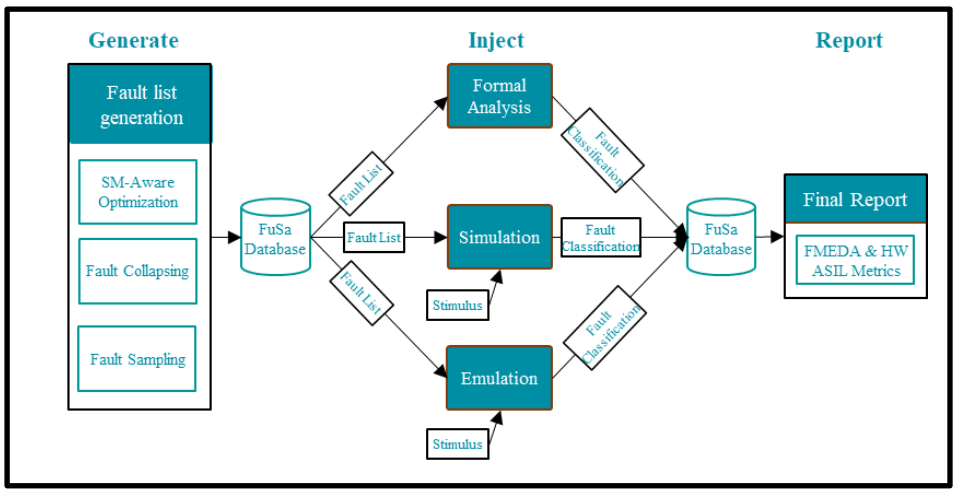

Figure 3: Efficient three step workflow with a common database

While these three steps are independent in execution, they need to be integrated to reduce the overall execution time. We achieved this integration and efficiency through a highly configurable common database. The common database supports directly reading the specific fault nodes per failure mode, using an optimal technique for fault injection, and then updating the fault resolution to calculate the FMEDA metrics.

V. Experiment on Automotive SoC

In the experimental work we did for this paper, we used a SoC Design with approximately three million gates, and we used both simulation and emulation fault injection engines to efficiently complete the fault campaigns for final metrics. Doing formal analysis is our next step as a part of finishing the overall fault injection on this automotive SoC. Figure 4 is a high-level block representation of the automotive SoC. The highlighted block, *safety island*, is the focus for this section of the paper.

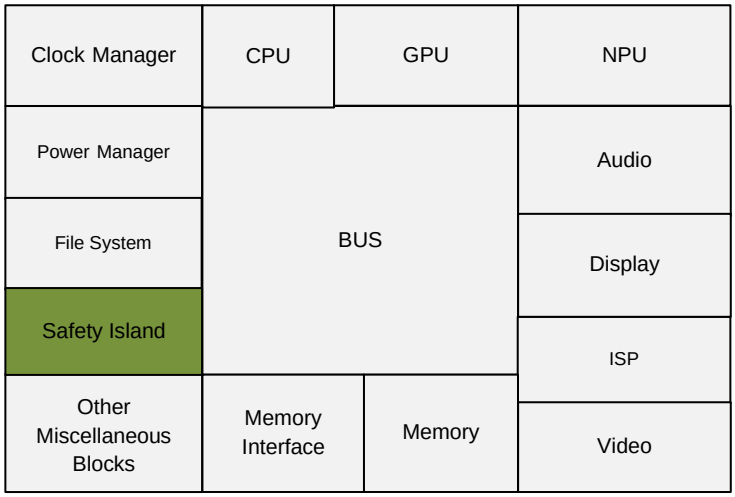

Figure 4: Automotive SoC top-level block diagram

Figure 5 is a representation of the safety island block from figure 4. The color-coded areas show where simulation, emulation, and formal engines were used for fault injection and fault classification.

Fault injection using simulation was too time and resource consuming for the CPU core and cache memory blocks. Those blocks were targeted for fault injection with an emulation engine for efficiency. The CPU core is protected by a software test library (STL) and the cache memory is protected by ECC, as shown in Figure 5. The bus interface requires end-to-end protection where fault injection with simulation was determined to be efficient. The fault management unit was not part of this experiment. Fault injection for the fault management unit will be completed using formal technology as a next step.

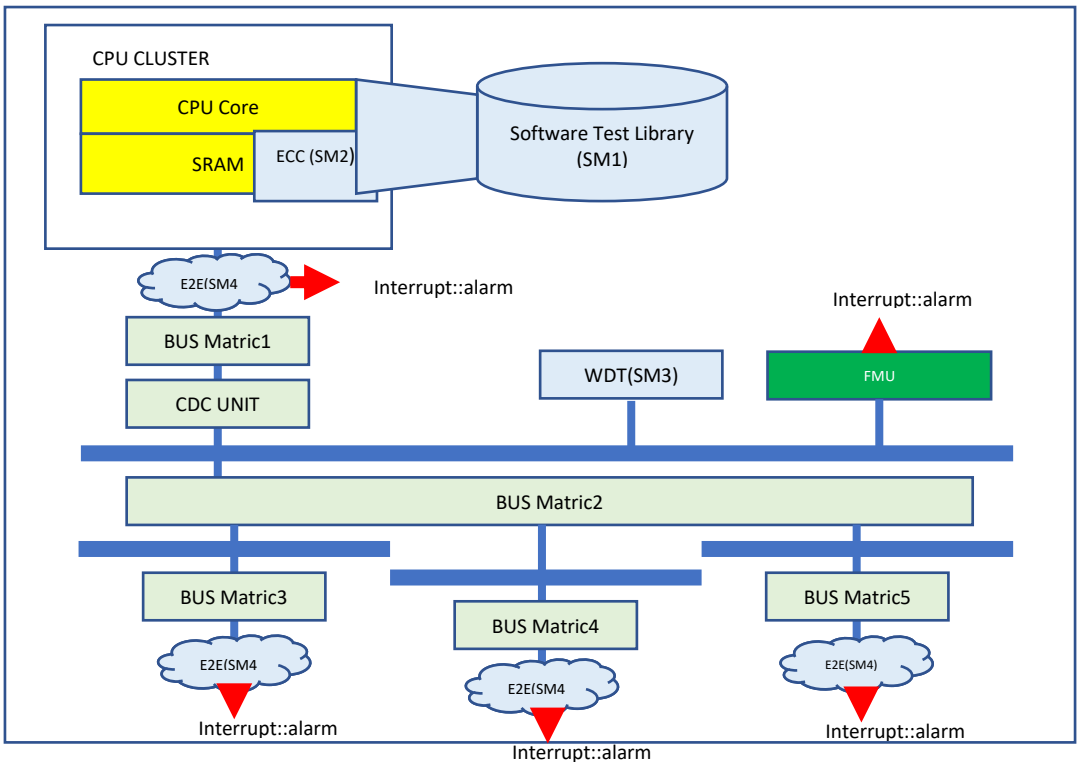

Figure 5: Detailed block diagram

Table 4 shows the register count for the blocks in the safety island. The fault lists generated for each of these blocks were optimized to focus on the safety critical nodes which have safety mechanisms/protection.

Block name	Register Count	Gate Count
Safety Island	890K	46.5 M
CPU Core + Cache mem	215K	7.1 M
Bus	470K	-
Fault Management unit + Glue logic	33K	-

Table 4: Block register count

SafetyScope, a safety analysis tool, was run to create the fault lists for the FMs for both the Veloce Fault App (fault emulator) and the fault simulator and wrote the fault lists to the functional safety (FuSa) database.

For the CPU and cache memory blocks, the emulator inputs the synthesized blocks and fault injection/fault detection nets (FIN/FDN). Next it executed the stimulus and captured the states of all the FDNs. The states were saved and used as a "gold" reference for comparison against fault inject runs. For each fault listed in the optimized fault list, the faulty behavior was emulated, and the FDNs were compared against the reference values generated during the golden run, and the results were classified and updated in the fault database with attributes.

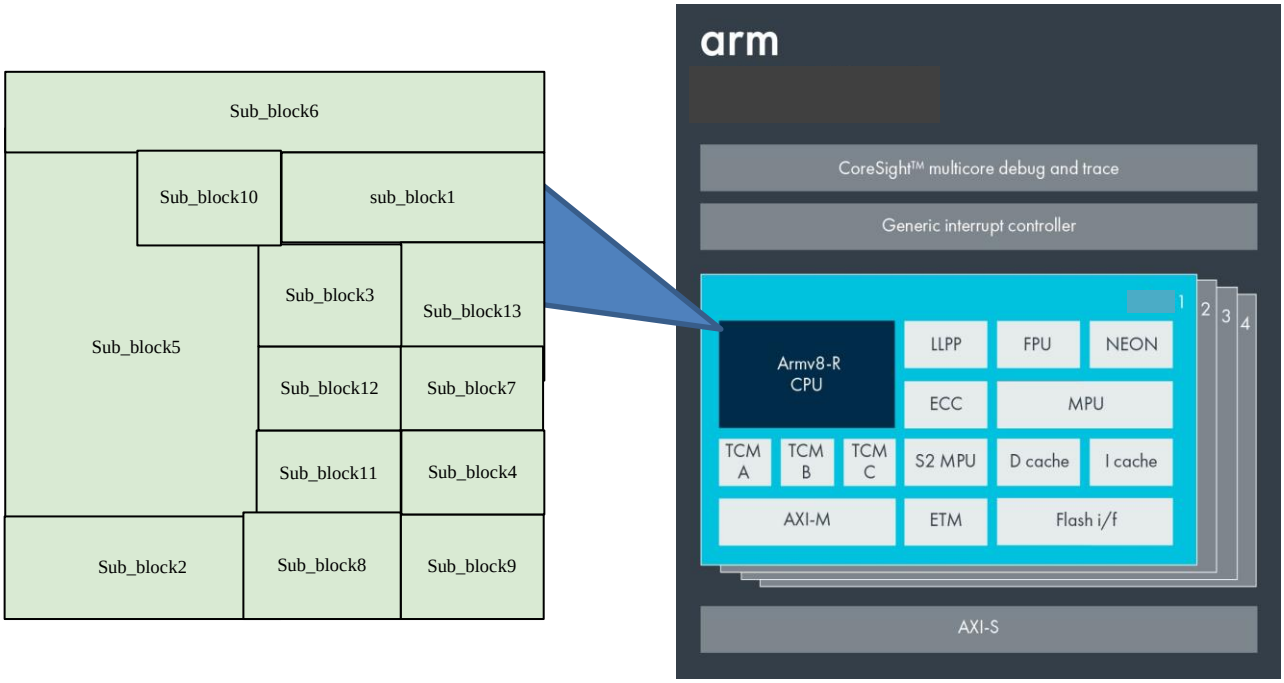

Figure 6: CPU Cluster
(*Source from https://developer.arm.com/Processors/Cortex-R52)

For each of the sub parts shown in the block diagram, we generated an optimized fault list using the analysis engine. The fault lists are saved into individual session in the FuSa database. We used the statistical random sampling on the overall faults to generate the random sample from the FuSa database. In the next section we will discuss in detail of taking 1 such random sample all the way through the fault injection using emulation. However, for this particular to completely close on the fault injection, we processed N samples.

Safety Mechanisms	Detected Fault Distribution (%)
ECC Correctable (SM 2)	16.86
ECC UnCorrectable (SM 2)	0.74
STL (SM 1)	80.64
WDT (SM 3)	1.76
Sub total	100.00

Table 5: Detected Faults by Safety Mechanisms

CPU Core + Cache Ram	Total Faults (SA0 + SA1)	Total Fault Distribution (%)	Sampled faults (SA0 + SA1)	Sampled Fault Distribution (%)	Test 1: Detected fault (%)	Detected fault Ratio (%)
sub_block1	85,860	13.50	604	12.63	315	52.15
sub_block2	4,738	0.74	40	0.84	0	0.00
sub_block3	46,194	7.26	332	6.94	162	48.80
sub_block4	22,828	3.59	202	4.22	135	66.83
sub_block5	315,982	49.68	2,350	49.14	1,719	73.15
sub_block6	6,664	1.05	48	1.00	0	0.00
sub_block7	19,202	3.02	144	3.01	97	67.36
sub_block8	8,084	1.27	58	1.21	17	29.31
sub_block9	8,100	1.27	66	1.38	39	59.09
sub_block10	49,292	7.75	426	8.91	305	71.60
sub_block11	9,306	1.46	46	0.96	31	67.39
sub_block12	34,832	5.48	286	5.98	218	76.22
sub_block13	24,964	3.92	180	3.76	87	48.33
	636,046	**100**	**4,782**	**100.00**	**3,125**	**65.35**

Table 6: Results of Fault Injection in CPU and Cache Memory

Above table shows that the overall fault distribution for total faults is in line with the fault distribution of the random sampled faults. The table further captures the total detected faults of 3125 out of 4782 total fault. We were also able model the detected faults per sub part and also provide an overall detected fault ratio of 65.35%. Based on the faults in the random sample and our coverage goal of 90%, we calculated that the Margin of Error (MOE) is ±1.19%.

The total detected (observed + unobserved) 3125 faults provide a clear fault classification. The Undetected Observed also provide a clear classification for Residual faults. We did further analysis of Undetected Unobserved and Not Injected faults.

Fault Classification	Total Faults (SA0 + SA1)	Total Fault Distribution (%)
Detected Observed	3125	63.35
Detected Unobserved	0	0.00
Undetected Observed	79	1.65
Undetected Unobserved	616	12.88
Safe Fault Dead Logic	234	4.89
Not Injected	728	15.22
sub total	4782	100.00

Table 7: Fault Classification after Fault Injection

We used many debug techniques to analyze the 616 Undetected Unobserved faults. First, we used formal analysis to check the Cone of Influence (COI) of these UU faults. The faults which were outside the COI were deemed safe and also there were 5 faults which were further dropped from analysis. For the faults which were inside the COI, we used engineering judgment with justification of various configurations like, ECC, timer, flash mem related etc. Finally using formal and engineering judgment we were able to further classify 616 UU faults into Safe faults and remaining UU faults into conservatively Residual faults. We also reviewed the 79 residual faults and were able to classify 10 faults into Safe faults. The Not injected faults were also tested against the simulation model to check if any further stimulus is able to inject those faults. Since no stimulus was able to inject these faults, we decided to drop these faults from our consideration and against the Margin of Error accordingly. With this change our new MOE is ±1.293%.

Final Result	Total Faults (SA0 + SA1)	Total Fault Distribution (%)
Detected Observed	3125	77.18
Detected Unobserved	0	0.00
Undetected Observed (Residual) 10 fault moved to SAFE Fault	69	1.70
Undetected Unobserved (conservatively Residual)	280	6.92
Safe Fault Dead Logic	234	5.78
Safe Fault Formal COI analysis Signal Back Propagation Safe Fault Configuration Engineer's justification	341	8.42
sub total 4782 – 733 (Not Injected)	4049	100.00

Table 8: Final fault classification post analysis

In parallel, the fault simulator pulled the optimized fault lists for the failure modes of the bus block and ran fault simulations using stimulus from functional verification. The initial set of stimuli didn't provide enough coverage, so higher quality stimuli (test vectors) were prepared, and additional fault campaigns were run on the new stimuli. All the fault classifications were written into the FuSa database. All runs were parallel and concurrent for overall efficiency and high performance.

BUS Related Logic	Total Faults (SA0 + SA1) %	Test 1: Detected fault (%)	Test 2: Detected fault (%)	Test 3: Detected fault (%)	Test 4: Detected fault (%)
NOC in Safety Island	100%	3.27%	5.50%	9.80%	TBD (Future work)

Table 9: Percentage of Detected Faults for BUS Block by E2E SM

Safety analysis using SafetyScope helped to provide more accuracy and reduce the iteration of fault simulation. CPU and cache mem after emulation on various tests resulted an overall SPFM of over 90% as shown in Table 5. At this time not all the tests for BUS block (End to End protection) doing the fault simulation have been completed. Table 6 shows the first initial test was able to resolve the 9.8% faults very quickly. We are integrating more tests which have high traffic on the BUS to mimic the runtime operation state of the SoC. The results of these independent fault injections (simulation and emulation) were combined for calculating the final metrics on the above blocks, with the results shown in Table 7. Execution and closer of the faults using Simulation is our future work.

Design Cpu + Cache mem	**Final SPFM %**
DCPerm	91.3805878
MOE	±1.293%

Table 10: Overall results

VI. Conclusion

An exhaustive approach to achieving and validating an optimal safety architecture is inefficient for large designs with complicated safety mechanisms. In addition, the large number of faults that need to be analyzed make performing safety verification using a single technology impractical. Using this SoC level test case, we demonstrated how interoperability of fault injection engines, optimization techniques, and an automated flow can effectively reduce overall execution time to quickly close-the-loop from safety analysis to safety certification. Close engagement between product teams, methodology teams and EDA vendors is equally critical as tools, methods and technique are evolving at a rapid pace. Advanced methodologies such as safety Analysis for optimization and fault pruning, concurrent fault simulation, fault emulation, and formal based analysis are deployed in this project to validate the safety requirements for the Automotive SoC. Performing safety analysis prior to running the fault injection is very critical and time saving. Therefore, as demonstrated in the paper the interoperability for using multiple engines and reading the results from a common FuSa database is necessary for a project of this scale.

REFERENCES

[1] Methodology for Efficient closure to Fault injection, Dvcon Europe 2022
[2] ISO 26262 Part 5: Product development at the hardware level, Second edition 2018-12
[3] ISO 26262 Part 11: Guidelines on application of ISO26262 to semiconductors, Second edition 2018-12
[4] Formal Techniques for Optimizing ISO 26262 Fault Analysis, Siemens Verification Academy
[5] Rambus RT-640 road to ISO26262 certification, Rambus white paper
[6] Validating the complex safety mechanisms, Siemens Verification Academy