Whose Fault Is It Formally? Formal Techniques for Optimizing ISO 26262 Fault Analysis.

Ping Yeung
Mentor Graphics, Fremont
ping_yeung@mentor.com

Doug Smith
Mentor Graphics, Austin
doug_smith@mentor.com

Abdelouahab Ayari
Mentor Graphics, Munich
abdelouahab_ayari@mentor.com

ABSTRACT

ISO 26262 states that hardware architectural metrics are required to assess the adequacy of safety mechanisms and their ability to detect and/or prevent faults from reaching safety critical areas. Fault injection is an essential method to determine the completeness and correctness of the safety mechanisms in meeting functional safety requirements. However, random fault injection is time-consuming and inefficient in meeting today's product release cycles. In this paper, we present a static analysis step to improve the efficiency of fault injection. In addition, a formal methodology is used for fault injection to maximize the observability of the safety failures.

I. INTRODUCTION

The automotive safety standard, ISO 26262 [1], states that safety analyses on hardware designs should include Failure Mode and Effects Analysis (FMEA). Hardware architectural metrics are required to assess the adequacy of the safety mechanisms and their ability to prevent faults from reaching safety critical areas. A process of fault analysis that includes fault injection is crucial for measuring and verifying the assumptions of the FMEA.

In Part 5 of the ISO 26262 [2] specification, a hardware architecture needs to be evaluated against the requirements for fault handling. It requires that the probabilities of random hardware failures are rigorously analyzed and quantified via a set of objective metrics [3]. If any of the architectural metrics fail to meet the criteria defined for the product's Automotive Safety Integrity Level (ASIL), re-evaluation of the component safety concept, improvement of existing safety mechanisms, and/or introduction of new safety mechanisms are mandated. Fault analysis is a process that aims to determine the percentage of faults that turn into failures that impact the hardware's safety requirements. It uses fault injection with design representations at different levels to compute and to verify the values of failure mode and diagnostic coverage. Hence, fault injection is an essential method to determine the completeness and correctness of the safety mechanisms in meeting hardware safety requirements.

Traditionally, fault injection involves running a functional simulation on the gate-level representation, flipping a bit at some point in time, then analyzing the results to see if any of the signals defined as safety critical (such as output ports, state-machines) had changed in any way. Determining the percentage of faults that will turn into failures requires injecting tens of thousands of faults into hundreds of tests, simulating them to their outputs, and comparing the results. However, a gate-level design representation is only available late in the design cycle. At that stage, any conceptual re-evaluation and/or architectural change will have a significant impact on the project schedule and is costly to perform. As a result, fault injections at the register transfer level [3] or even at the transaction level [4] are important alternatives to perform fault analysis and to assess the efficiency of the safety mechanisms early in the design cycle.

Even at a higher level of abstraction, random fault injection is still time-consuming and inefficient in meeting today's project schedules. Hardware emulation or design prototyping is useful for addressing the performance bottleneck in simulation-based fault analysis. Yet these approaches are inefficient as the injected faults may not cause any failure of the safety goals (safe faults) or the failures may be prevented by the safety mechanisms. To be efficient, we want to focus on faults that are not guarded by any safety mechanism and will subsequently cause failures of the safety goals (single-point faults, residual faults).

These factors motivate us to adopt formal verification for fault analysis. In this paper, we present a static analysis step to derive, optimize, and prioritize the fault list so that it can be analyzed as efficiently as possible. Also, a formal methodology is used for fault injection to maximize the observability of the faulty results.

II. FORMAL FAULT PRUNING

We want to improve the efficiency of any fault injection mechanism. To do so, static analysis is performed ahead of time to determine the critical set of design elements where faults should be injected. At the same time, we can remove design elements that are not critical to the safety mechanism. We call this process *fault pruning* [6]. The primary objective is to skip elements that will not affect the safety requirement and focus fault injection on elements that will.

A. Fault pruning based on safety goals

As part of the ISO 26262 safety analysis and based on the safety requirement, an engineer defines the safety goals and the safety critical elements for a design. A formal tool has the ability to trace back from the safety goals through the design elements to determine what is in the cone of influence (COI), Figure 1.

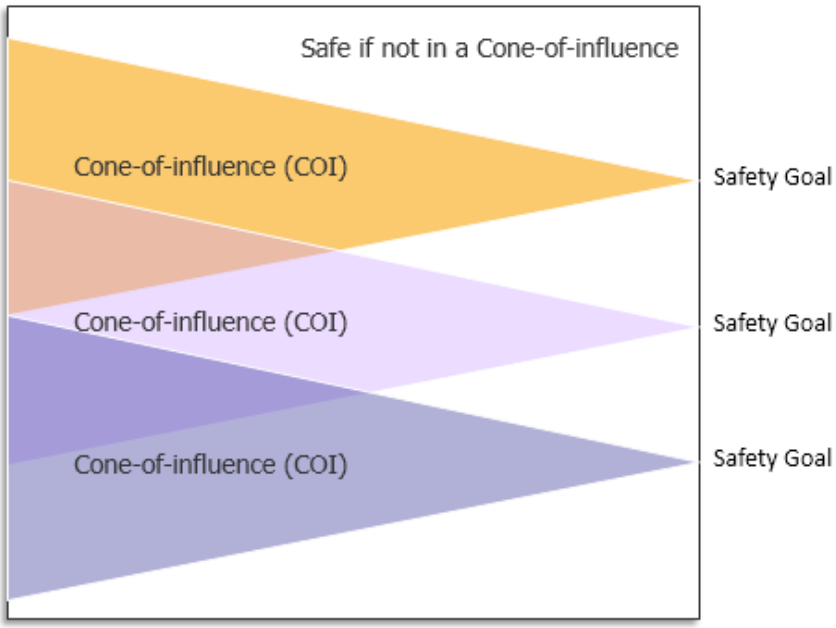

Figure 1: Formal tools trace back the cone of influence to perform their analysis.

The steps of the fault pruning process can be summarized as:

1. Identify a set of safety critical elements in the design that will have an immediate impact on the safety goals. These include output ports, state-machines, counters, configuration/status registers, FIFOs, and other user-specified elements.
2. Create a list of constraints for the block. Examples can include debugging mode, test mode, and operational modes not used in a specific instance, etc. However, for this list of constraints to be valid, these constraints must have an independent safety mechanism. For example, an error flag that is generated whenever the design sees these modes are active. Typically these modes are global and their safety mechanisms are addressed at higher levels.
3. Compute the cone of influence (COI) of these safety critical elements and create a fault injection list based on elements in these fan-in cones. Design elements that are not in the cone of influence are considered safe and do not need to be considered.
4. Apply formal techniques to verify that faults injected in these elements will be observed and will impact the safety goals and//or safety critical elements.

B. Fault pruning based on safety goals and safety mechanisms

The next step is to look at what design elements in the COI of a *safety requirement* overlap with the COI of the *safety detection mechanism*. For example, suppose the data integrity of a storage and transfer unit is considered safety critical. As shown in Figure 2, along with each data packet, there are error-correcting code (ECC) bits for detecting any error. As long as the ECC bits can flag any transient fault, the data output is protected because errors can be detected and dealt with downstream. However, in a dual-point fault situation, error(s) may also be affecting the ECC detection logic. As a result, the safety mechanism is prevented from detecting bad data in the packet. In this case, the first fault in the safety mechanism is considered a masking condition and thus a latent fault. The second fault in the dual-point fault scenario is an actual error that the safety mechanism would have detected if not for the latent fault, thus leading to a dual-point failure.

Besides, not all design elements are in the COI of the safety mechanism. Design elements that are in the COI of the data output but not in the COI of the ECC logic will not be protected by the safety mechanism and so must be considered potentially unsafe. Faults in these design elements are treated as undetectable or potentially unsafe. As a result, we have:

1. Design elements (in the COI of the ECC) where faults are potentially detectable by the safety mechanism (safe, residual faults, or dual-point faults)
2. Design elements (out of the COI of the ECC) where faults are undetectable by the safety mechanism and potentially unsafe (residual faults)

With the ability to measure the two COI, the opportunity to improve the design (hardening) becomes apparent. A large "undetectable" section of the two cones means that there is a higher percentage of elements that can lead to residual faults, and the ability to meet high ASIL ratings is significantly reduced. The goal of the design team at this point is to determine how to create more overlap of the safety critical function and the safety mechanism.

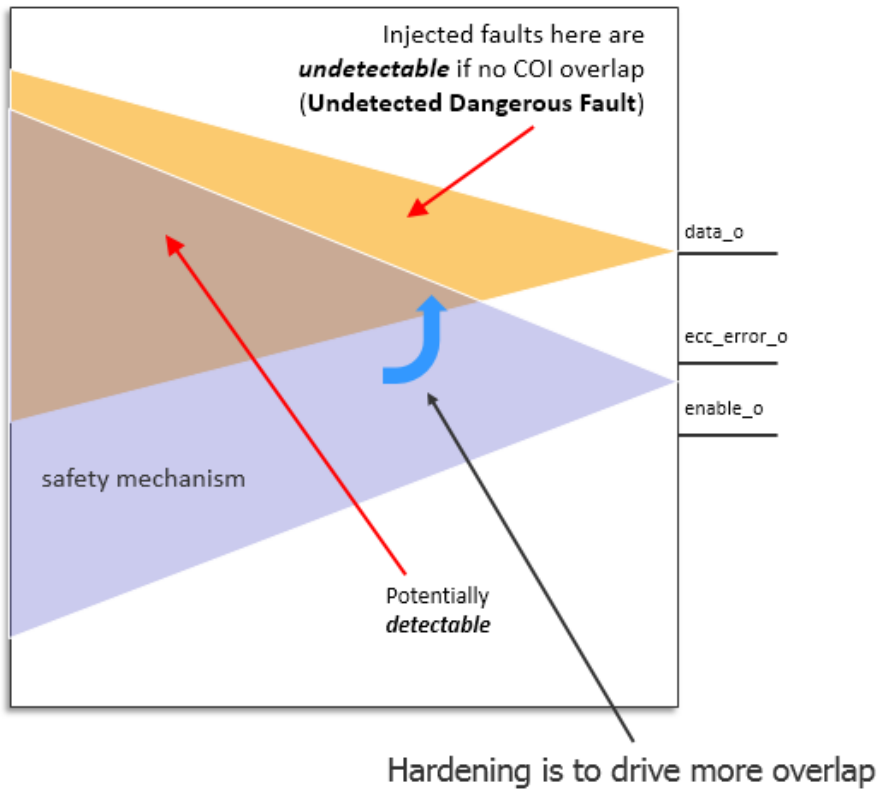

Figure 2: Anything outside the COI of the detection logic is considered potentially unsafe.

III. Formal Fault Injection

Once the number of design elements in the fault list have been pruned down to a high quality and meaningful subset, it can be passed onto various fault injection mechanisms; such as fault simulation, hardware accelerated fault simulation, formal fault verification, and hardware prototyping. Although simulation-based fault injection approaches, such as software and hardware accelerated fault simulation, are common, they are known to be inefficient.

A. Formal-based fault injections

Even though most simulation regression environments deliver high functional coverage, they are poor at propagating faults to the output ports, and the functional checks are not robust enough to detect all kinds of faults. We create the regression environment to verify the functional specification, i.e., the positive behaviors of the design. It is not designed to test the negative behaviors. We are interested in formal-based fault injections primarily because of these two attributes.

1. Fault propagation: For instance, a random fault has been injected into the FIFO registers that causes its content to be corrupted. As the simulation environment is not reactive, the fault can be overwritten by subsequent data being pushed into the FIFO. Formal verification is different; it injects faults intelligently. It is good at introducing faults that it knows how to propagate to affect the safety goals or outputs of the design.
2. Fault detection: For instance, even when the corrupted FIFO content has been read, the fault may not be detected. The FIFO content may be discarded because the functional simulation is not in the mode to process the data. Most functional simulations test the scenarios that have been defined in the specification, while fault detection requires testing all the scenarios that may go wrong (negative behavior) consistently. To check all potential negative scenarios, we rely on formal sequential equivalency checking.

B. Fault injection with sequential logic equivalency checking

Formal Sequential Equivalency Checking (FSEC) compares the outputs of two designs or two representations of the same design [5]. The implementation of the two designs can be different as long as the outputs are always the same. For example, FSEC can be used to compare a VHDL design that has been ported to Verilog (or vice versa) to see if the two RTL designs are functionally equivalent. Conceptually, an FSEC tool is a formal tool with two designs instantiated, constraining the inputs to be the same, and with assertions specifying the outputs should be equivalent for all possible internal states.

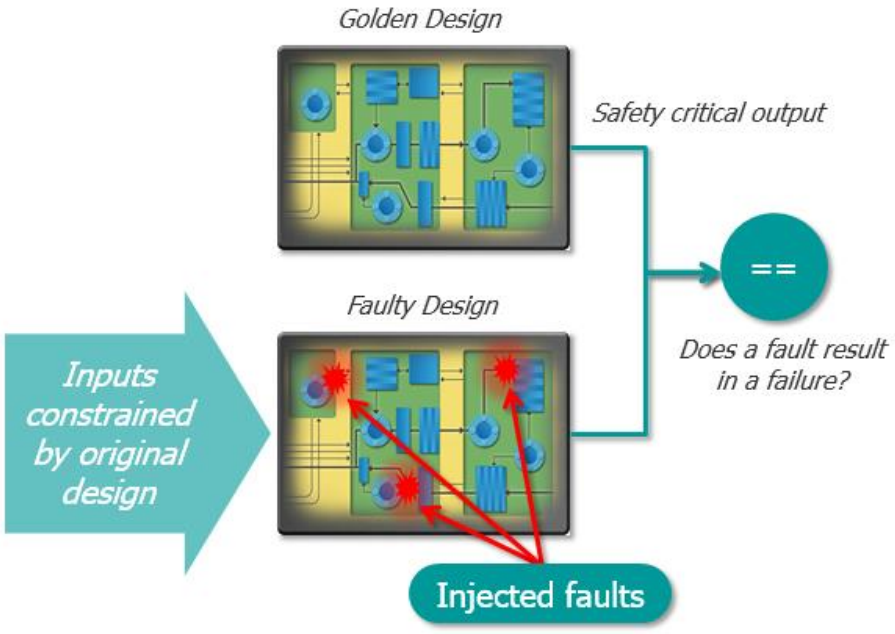

Figure 4: Fault injection with sequential logic equivalency checking

While there are many uses for FSEC, fault injection is a particular sweet spot. Formal tools have the ability to inject both stuck-at and transient faults into a design, clock the fault through the design's state space, and see if the fault is propagated, masked, or detected by a safety mechanism. Once the fault is injected, the formal tool tests all possible input combinations to propagate the fault through the design to the safety mechanism — this is where FSEC becomes necessary [6].

As depicted in Figure 4, a golden (no fault) model and a fault injected model are used to perform on-the-fly fault injection and result analysis. Fault injection with formal FSEC is handy early in the design cycle when no complete functional regression is available. It can be used to perform FMEA [7] and to determine whether the safety mechanism is sufficient [8]. Designers can experiment and trade off different safety approaches — including error detection, error correction, redundancy mechanism, register hardening, and so on — quickly at the RTL.

By instantiating a design with a copy of itself, all legal input values are automatically specified for FSEC, just as a golden reference model in simulation predicts all expected outputs for any input stimulus. The only possible inputs are those values that can legally propagate through the reference design to the corresponding output. By comparing a fault injected design with a copy of itself without faults, the formal tool checks if there is any possible way for the fault to either escape to the outputs or go undetected by the safety mechanism.

The steps of the fault injection with FSEC can be summarized as:

1. Specify two copies of the original design for FSEC. The input ports will be constrained together, and the output comparison will be checked automatically.
2. Run FSEC to identify any design elements — such as memory black boxes, unconnected wires, un-resettable registers, etc. — that will cause the outputs to be different. Constraints are added to synchronize them.
3. Use the fault list to automatically specify possible injection points in the faulty design. Normally, we will focus on the single point faults; i.e., one fault will be injected to check for equivalency.
4. Based on the fault list and the comparison points, FSEC will automatically identify and remove any redundant blocks and logic.
5. We run FSEC concurrently on multiple servers. The multi-core approach has significant performance advantage as multiple outputs can be checked on multiple cores concurrently.
6. We use a technology to compile all the faults in one single run and thus save time and space resources usually needed for compiling the faults one by one.
7. We also run the FSEC verification step for thousands of faults in one single run.
8. If an injected fault has caused a failure at the comparison point, a waveform of the counter-example that captures the fault injection and propagation sequence can be generated for debugging.

IV. RESULTS

We have put together an environment for formal fault pruning and fault injection with FSEC as summarized in figure 5 below. The fault pruning step builds a netlist representation of the design. Based on the safety goals (that may be just the output ports) and the safety mechanisms, a formal method is then used to compute and examine the cone of influences. Safe and unreachable faults are reported. Fault injection with FSEC is done using the Questa® Formal Verification tool [5]. Before running fault injection, users can control the locations and the types of faults to be injected using the fault modeling and constraints. In addition, users can partition the design or configure the tool to perform the injection runs on a multi-core server farm environment.

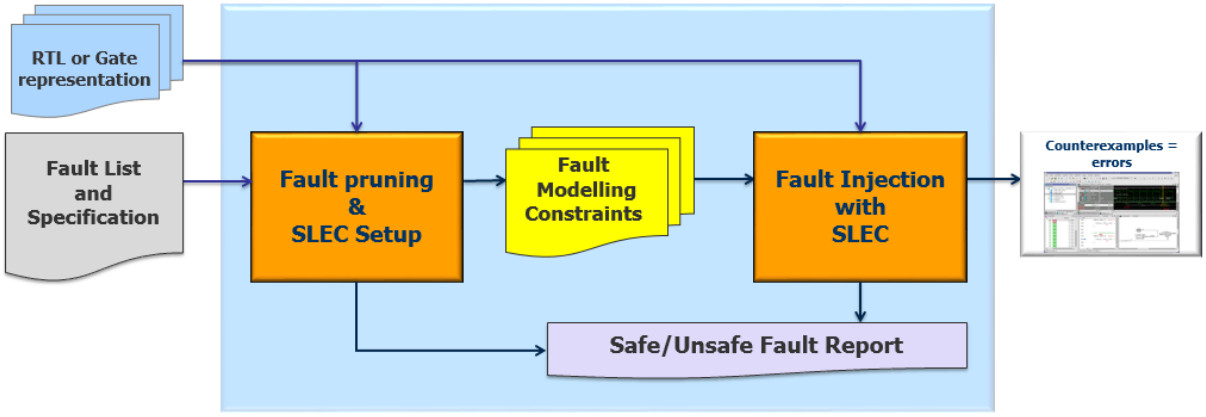

Figure 5: Fault pruning and injection environment

A. Results of formal fault pruning

Case 1: The design is a float point unit with approximately 530K gates. The goal is to identify the *safe* faults in the design. These faults are outside the COI or cannot be propagated to functional outputs regardless of input stimuli. The original fault list contains approximately 32K faults. After compilation and set up, 868 of them were identified as *safe* faults. Interestingly, due to the mode of operation, 690 of them cannot be propagated and will not affect the outputs of the design.

Case 2: The design is a memory management unit with approximately 1.3M gates. The goal is to identify faults that can propagate to internal *status* registers. These registers are checked by safety mechanisms at a higher level. Hence, these *detectable* faults can be considered safe. In the design, there are 71 *status* registers and 1524 faults in the targeted fault list. After compilation and set up, 720 of them were identified as *safe* faults within an hour.

Case	Gates	Faults	Safety Mechanism	Run Time per Fault	Safe Faults	% Safe Faults
#1	530K	32425	0	3.6 sec	868	2.7%
#2	1300K	1524	71	2.3 sec	720	47%

B. Results of formal fault injection

Case 1: The design is a clock controller block with a safety mechanism implemented with triple modular redundancy (TMR). TMR is an expensive, but robust, on-the-fly safety mechanism. For a quick preliminary test, faults were allowed to be injected to all the registers in the design (Case 1a). As all the registers are in the COI of the safety mechanisms, there is no surprise. Formal fault injection verifies that all the injected faults will be caught by the safety mechanisms. Next, faults were allowed to be injected to all the nodes (registers, gates, and wires) in the design; there are significantly more faults (Case 1b). Again, formal fault injection verifies that all the injected single point faults will be guarded by the safety mechanisms.

Case	Faults	Number of Faults	Run Time	% Missed by Safety Mechanisms
#1a	registers	57	15 min	0
#1b	all nodes	2648	265 min	0
#2a	registers	267	23 min	12%
#2b	all nodes	12963	1332 min	8%

Case 2: The design is a bridge controller that consists of the clock controller block from Case 1. In this case, formal fault injection was able to inject and propagate some faults to the output ports of the design. Two types of faulty scenarios were observed:

- Single point faults that were not protected by any safety mechanism
- Residual faults that were protected by safety mechanisms; however, the safety mechanisms did not detect the error conditions correctly.

Figure 8 shows a scenario of Case 2 where the injected fault was missed by the safety mechanism and violated the safety goal. A random fault was injected shortly after reset removal. This fault had caused multiple errors in the design, and they reached the outputs of the design after five clock cycles. Unfortunately, the safety mechanism was not able to detect these errors. As a result, the fault has caused a violation of the safety goal.

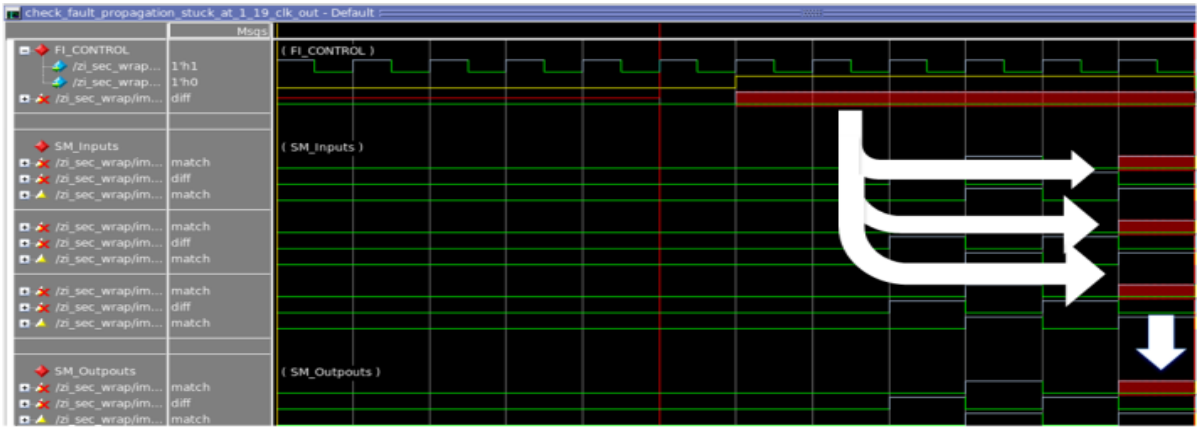

Figure 8 Waveform showing how faults are propagated to outside safety mechanism

V. SUMMARY

In this paper, we have discussed formal fault pruning and fault injection with formal sequential equivalency checking. By leveraging formal technology, we can determine the number of safe faults by identifying the unreachable design elements, those outside the cone of influence (COI), or those that do not affect the outputs (or are gated by a safety mechanism). After fault pruning, the optimized fault list can be used for fault injection using various mechanisms; such as fault simulation, hardware accelerated fault simulation, formal fault verification, and hardware prototyping. Simulation-based fault injection is not efficient in propagating faults to the output ports, and the traditional functional checks are not robust enough to detect all kinds of faults. With simulation, it is never known if the design has been simulated enough or given enough input scenarios. Formal verification conclusively determines if faults are safe or not, making the failure rates from formal analysis more comprehensive than fault simulation.

REFERENCES

[1] Barbara J. Czerny, Joseph D'Ambrosio, Rami Debouk, Kelly Stashko, General Motors, *ISO 26262 Functional Safety Draft International Standard for Road Vehicles: Background, Status, and Overview*, 28th International System Safety Conference, 2010.

[2] ISO 26262-5:2011 Road vehicles Functional safety, Part 5: Product development at the hardware level, www.iso.org.

[3] Avidan Efody, Mentor Graphics Corp, *Whose Fault Is It? Advanced Techniques for Optimizing ISO 26262 Fault Analysis*. DVCon 2016.

[4] Bogdan-Andrei, Tabacaru, Moomen Chaari, Wolfgang Ecker, Thomas Kruse, and Cristiano Novello, Infineon Technologies AG, *Fault-Effect Analysis on Multiple Abstraction Levels in Hardware Modeling*, DVCon 2016.

[5] Questa Formal Verification Apps. https://www.mentor.com/products/fv/questa-formal-verification-apps

[6] Doug Smith, Mentor Graphics Corp, *How Formal Reduces Fault Analysis for ISO 26262*, Mentor Graphics white paper, www.mentor.com.

[7] Riccardo Mariani, Gabriele Boschi, Federico Colucci, Yogitech SpA, *Using an innovative SoC-level FMEA methodology to design in compliance with IEC61508*, DATE 2007.

[8] Adrian Traskov, Thorsten Ehrenberg, Sacha Loitz, Continental Teves AG, Abdelouahab Ayari, Avidan Efody, Joseph Hupcey, Mentor Graphics Corp, *Fault Proof: Using Formal Techniques for Safety Verification and Fault Analysis*, DVCon Europe 2016.