2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Fault Proof: Using Formal Techniques for Safety Verification and Fault Analysis

Adrian Traskov,Thorsten Ehrenberg, Sacha Loitz

Continental Teves AG & Co. oHG, Frankfurt a.M., Germany

Abdelouahab Ayari, Avidan Efody, Joseph Hupcey III

Mentor Graphics

***Abstract*— Safety mechanisms designed to correct/detect random hardware failures implement critical functionality but are relatively low in gate count, which often makes them an ideal application for formal verification techniques. In this paper we present a case study describing fault analysis of a Triple Modular Redundancy (TMR) element and its associated majority voter using formal. We start by describing a generic flow for fault analysis of safety mechanisms, including fault population reduction, fault injection, checking and classification, and collection of metrics. We then move on to show how formal can be used to perform each of these tasks in the context of a TMR safety mechanism. Finally we compare formal results and run time against those obtained using dynamic simulation techniques, and show how formal is able to minimize the analysis effort required.**

Keywords*— *ISO 26262; fault analysis; formal verification, property checking, single event upset, formal verification of safety mechanism

I. INTRODUCTION

One of the cornerstones of ISO26262-compliant design is the inclusion of additional logic to implement a "safety mechanism", whose purpose is to ultimately guarantee safe behavior of the given system in the face of expected failure modes. Specifically, this hardware must detect enough faults to meet the target ASIL (Automotive Safety Integrity Level) requirements, and provide a deterministic and "safe" reaction to the faults – whether it is a correction and recovery from of a given fault, or a transitioning of the system into a safe state. Naturally, thorough verification of this critical piece of logic in the face of all potential faults in the system, and in the safety mechanism logic itself, is required.

Since the early days of fault verification (with DO-254 and other milaero safety standards that predate ISO26262 [1]), simulation-based approaches have dominated the landscape of verification methodologies applied to this challenge. However, even the most well-designed testbenches are not "exhaustive" – they leave numerous state spaces and control logic combinations unverified; let alone consider all the new states that are introduced by random hardware faults. Conversely, because formal verification tools develop a mathematical model of the given device under verification (DUV), it is possible to exhaustively verify all inputs and outputs against the expected behaviors of all states in a DUV. In the past such analysis could only be done on very small DUVs. Today, advances in formal algorithms and the corresponding "engine" implementations make it possible to verify multi-million gates of logic, and in some cases whole systems, depending on the specific verification requirement, enabling abstractions, and fault population optimizations that the DUV's architecture implicitly supports.

II. FAULT MODELS

Though the effects of aging and radiation could make real hardware fail in numerous different ways, each with its own specific characteristics, at an RTL or gate level (GL) of abstraction, these failures would usually be modeled in one of the three ways below:

- **Stuck-at faults –** a given net or register are constantly tied to 0 or 1 either from time 0 (lower ASIL) or from an arbitrary time (higher ASIL)
- **Transient faults –** a given net assumes a constant 0 or 1 value for an arbitrary number of cycles, and then goes back to correct behavior
- **Bridging –** A given net assumes the value of another net or two nets assume a value that is the logical function of both of them (wired-and, wired-or)

III. SAFETY MECHANISMS

Some of the most common safety mechanisms used to detect and correct the above mentioned faults are finite state machine (FSM) with Safe Encoding, Triple Modular Redundancy (TMR), Error Correction Code (ECC), and "Lock Step" These safety mechanisms are taken from the relevant section of ISO-5, and our own experience. An in depth review of each of these is beyond the scope of this paper, but the key verification challenges posed by all of these methods are that:

- The state space for each can grow rapidly depending on the corresponding circuit being protected
- Because this "extra" logic can have a significant impact in area and power consumption, designers can be tempted to over-optimize it, which can give rise to "bugs of omission".

As such, any verification methodology and tool chain must be able to handle this.

IV. VALIDATION OF SAFETY MECHANISMS

Recall that the purpose of the safety mechanism is to detect enough faults for the target ASIL, and provide a deterministic and "safe" reaction to diagnosed faults – whether it's a correction and recovery from of a given fault, or transitioning the system into a safe state. Hence, there are two aspects that need to be verified:

- **Functional behavior** – Verify that the safety mechanism's specification and underlying requirements are satisfied. For example: verify that the design behaves as expected without presence of faults, and then again only for a specified list of faults. Also confirm that the design goes into a safe state for a specified list of faults.
- **Diagnostic coverage** – Verify the chosen safety mechanism is able to detect and handle a high enough percentage of faults for the required ASIL.

To verify the functional behavior and diagnostic coverage of the safety mechanism, we propose the flow described in Figure 1 below. The flow starts from a **safety specification** and an RTL or GL model of the design. The safety specification includes details about the safety mechanisms implemented in the design, the safety critical areas (as a list of signals), and the required ASIL level. Based on this information it then automatically picks up the relevant fault models, as well as the right abstraction level for the design, and creates an optimized list of faults. These faults are then injected on one of a few possible engines and checked automatically wherever possible. Finally, functional coverage, diagnostic coverage and ISO 26262 metrics such as Single Point Fault Metrics (SPFM), Latent Fault Metrics (LFM) are calculated and reported.

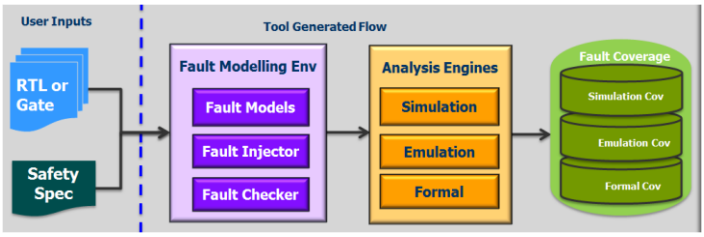


Figure 1 Fault verification flow block diagram

A. *Defining and Optimizing the "Fault Population"*

For anything beyond the simplest fault models (i.e. stuck at starting from time 0) and the simplest designs (a few thousand gates) fault population size will usually be an issue.

For dynamic methods (i.e. simulation, emulation, and FPGA prototyping) usually a statistical method is used to reduce the fault population size to allow it to be run in reasonable time by state of the art tools. The statistical method, however, has some problems, mainly because items in the population are not entirely independent of one another. Also, for areas that are highly critical, or for the logic of the safety mechanisms themselves, users and assessors might find the statistical approach problematic.

The advantage of formal methods is that they can check 100% of the faults on a given signal, which is something that dynamic method can almost never do. However, fault population size will still be an issue if the number of signals in the design is big. Hence it is best if the signal list is reduced to the minimum required either manually (by focusing the tool on critical areas) or automatically. Signals that could be usually removed automatically include:

1. Signals outside fan in of safety critical logic
2. Signals impacting safety critical logic only in non-operational nodes (debug/scan)
3. Nets equivalent to another net (under given input assumptions)
4. For GL: internal cell nets.

In addition, signals could be prioritized: for example, the impact of transient faults in logic is usually negligible and ISO guidelines don't require it even for highest ASIL unless shown to be relevant.

B. Fault Simulation – Necessary but not sufficent

Today, simulation is the de-facto verification approach used to determine whether a circuit is operating as expected. Simulation is used on a daily basis, is well understood, and is deeply integrated into most verification flows. Augmenting standard simulation with techniques and algorithms to model fault injection is the obvious way to develop new tools (i.e. fault simulators) for fault verification. Some companies have already produced formal-based solutions that work within fault simulation environments to accelerate and improve the process [4].

Indeed, fault simulation is relatively easy-to-use and provides clear diagnostic coverage. However, it suffers from three significant drawbacks. First, the fault population is very large, so the number of fault tests cases increases astronomically. Second, because simulation is incomplete by nature, the fault verification will be also incomplete. That is, faults could be not propagated to an output, and thus be easily observed would likely be missed by a simulation-based testbench. Third, whereas the modeling of some fault models (like stuck-at's) is simple and "almost" complete, the modeling of transient fault (SEU) is much more difficult. The critical point here is the time when the transient fault has to happen. The simulator will only insert the fault at a specific point in time, and so potentially missing points at which the fault can actually disrupt the circuit's operation (like a real world fault could). To illustrate this, consider the table below that shows impact of faults in simulation. We used a SPI Bridge [2] with an UVM testbench with 4 test cases and a popular industry RTL simulation tool (Mentor's QuestaSim). In our example here, we are considering only a small part of the design where faults are expected. Referring Table 1 the column "Fault Category" describes the kind of faults: any design nodes, only cell ports, only cell outputs, and storage elements. In the "Fault Coverage" column we are measuring the detected faults. We can conclude here that fault population reduction and optimization of the most important techniques both have a direct impact on fault coverage with a fault simulation.

Table 1 Fault Simulation Results for SPI Bridge

Fault Category	**Fault Simulation Regression Results**		
	Number of faults	***Run time***	***Fault Coverage***
No faults	0	00m:05s	NA
All nodes	2648	11m:44s	54,6%
No internal nodes	1428	06m:28s	52,5%
Cell outputs	449	01m:56s	54,0%
Storage elements	57	00m:32s	90,4%

C. A Formal-Based Approach – Exhaustive results

In recent years, formal verification has emerged as means to address the limitations of simulation: the ability to check only a small fraction of the behaviors of non-trivial hardware designs. As noted above, a formal-based approach develops a mathematical model of the given device under verification (DUV), so it is possible to exhaustively verify all inputs and outputs against the expected behaviors of all states in a DUV. Hence, formal verification can be employed to verify that a given design satisfies a given requirement as specified by industry

standard assertion languages that allow the D&V engineer to define all legal input patterns. Use of this approach is on the rise and a substantial amount of work has been done to characterize the verification problems that are optimal for formal verification. As a result, in recent years, more and more companies are deploying formal verification for components of designs that are difficult to verify sufficiently with dynamic simulation [5][6].

Nowadays the trend in formal verification is to develop automated formal applications -- tools that are built around the given verification task without requiring the user to know about formal itself, simplifying the workflow for all concerned. The formal-based approach we are advocating is very much along these lines – the goal is to automate as much as possible the user's workflow while simultaneously benefiting from the exhaustive nature of the formal analysis being done "under the hood".

1) Formal Technology

Fault Injection is a technique being applied in commercial formal solutions today, for use in safety critical automotive devices as well as other high reliability applications [7][11]. In our formal fault injection approach, we will use the sequential equivalence checking technology on the DUV's HDL. In short, sequential equivalence checking is an exhaustive comparison of both the logic of two circuits, and the temporal behavior of these circuits. There are two reasons for using sequential equivalence checking in our fault analysis approach: first, we can apply two copies of the design: one copy for fault injection another that is used as reference when checking for the impact of a fault injection. Second, because fault injection is done at an arbitrary time point, we have to guarantee that the signal in question has a design-value before and after the fault injection, and that this design value is captured from the reference design. Roughly speaking, we transformed the design into an XOR-Miter circuit [3] and its robustness against faults.

2) Formal Fault Injection Flow

The general flow is depicted in Figure 2. There are three inputs: the design under verification (equipped with a safety mechanism), a hierarchical link referencing the safety mechanism, and a set of fault points with their expected handling from the safety mechanism. Again, the flow uses two copies of the design: one copy of the design is for injecting faults, and the other copy of design is used as the reference. Furthermore, a module called fault injector controls injection and guarantee that the injected fault behaves exactly as the intended fault model.

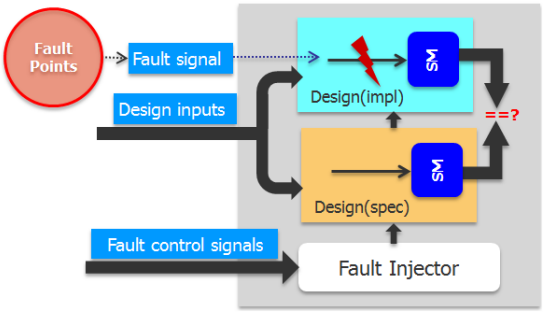


Figure 2 Formal Fault Injection Flow Based on Sequential Equivalence Checking

a) Design Under Verfication

Because we need to mimic physical faults that happen in a circuit, we are targeting gate-level design with balanced clocks and the scan-chain in-place. Generally, gate-level designs are not suitable for property checking and sequential analysis as formal model optimization becomes difficult in gate-level. But in this flow we are usually focusing on hardware components with relatively low number of gates. That said, it's interesting to note that this approach is applicable for either RTL or gate-level design descriptions. Typically, when this formal approach is applied at the RTL stage it is mainly used as a "bug hunting" tool to verify that the safety mechanism is working as expected. So much the better that circuit design changes at the RTL level are much easier for designers to implement and re-verify. In contrast, gate level formal verification is more of a final validation flow.

b) Specifying Safety Mechanism

The safety mechanism is the part of the design that is in charge of handling all the random faults that could occur during operation. This can be module instantiations, or a list of design signals partitioned as safety mechanism inputs and safety mechanism outputs. The modeling of various safety mechanism types such that their effectiveness may be verified using Formal tools is becoming common for automotive designs. For a thorough discussion of these mechanisms, see [8]. In Figure 3, we depicted the submodules of an SPI Bridge with a TMR safety mechanism. In this example, the three TMR instances will be specified as safety mechanisms. Having identified the design signals that are used as inputs (outputs respectively) for the safety mechanism, we can automatically generate SystemVerilog Assertion (SVA) properties to check propagation and detection of faults. Let $in_1, \ldots, in_n$ and $out_1, \ldots, out_m$ be the inputs and outputs of the safety mechanism. To check whether a fault is observed at the safety mechanism, we compare the inputs of the two copies of the safety mechanisms (instantiated in the two design copies). This is captured in following property.

$$fault\_injected\_at(sig)\ \#\#[0:\$] \bigcup_i (impl.sm\_inst.in_i == spec.sm\_inst.in_i)$$

We use the SVA delay operator as the fault can take time before being observed at the safety mechanism. To verify that for a particular fault, the safety mechanism forces the system to go to a safe state, we use following property where $state$ is the system state variable and $safe\_state\_value$ is the system state safe value.

$$fault\_injected\_at(sig)\ |\!\!-\!\!>\ \#\#[0:\$]\ (impl.state == safe\_state\_value)$$

To verify that a fault is detected by the safety mechanism and the safety mechanism "correct" the fault, we use following property

$$\bigcap_i fault\_injected\_at(sig)\ \rightarrow (impl.sm\_inst.out_i == spec.sm\_inst.out_i)$$

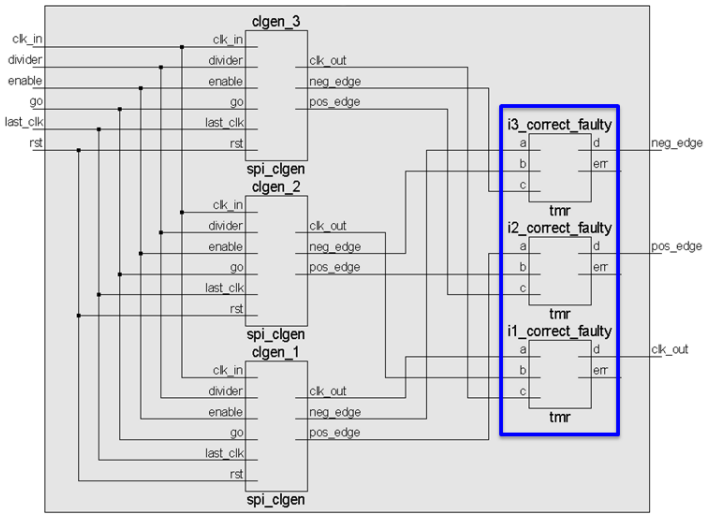


Figure 3 SPI Bridge with TMR

c) Fault Points Specification

The flow allows configuring the set of fault points with their corresponding fault models. In the example listed in Figure 4, all signals within the instance spi_clgen except all signals in the TMR instances are considered as fault points. Also in this example we are considering a transient fault (SEU), and expecting that all faults will be captured by the safety mechanism and handled accordingly.

```
#fault model {seu|s0|s1|bridging/open}
configure fault_model  seu

# In presence  of faults, system behaves  normal or goes to safe_state
# safe_state is a Boolean condition
#configure safe_state  {<some_sign> = = value}

# number of faults per run
configure fault_number 1

# fault Points
create fault top.spi_clgen.* –except top.spi_clgen.i_*_faulty
```

FaultCheck.do

Figure 4 Safety Configuration

d) Modeling Fault Injection in a Formal Verification Environment

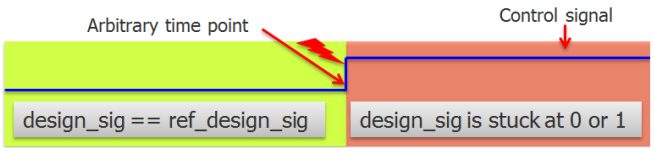
Modeling the most popular fault models introduced in Section III in a formal verification environment is a straightforward task. For earlier work on fault modeling in formal tools refer to [8], [9], [10]. Like fault simulation, this modeling will be achieved without instrumenting the design under verification. Modeling a specific fault for a given signal consists of two actions:

1. Prevent the signal to be driven from the design at a specific time point.
2. Force a specific value on the signal for a given period of time. The value to force and how long the force command has to take place depends on the fault model.

For permanent faults (stuck-at-0 and stuck-at-1), once the formal tool starts to inject a fault then the signal will be stuck-at 0 (or 1) for the rest of the run. In this case, we have only a start point of the fault injection but no end point (as it is not needed). However, for transient fault there is a start and end point for fault injection. We invoke fault injection at a time point – the time point when a non-design driven value is forced to the signal. To enable the formal tool to control when to inject faults, we will use a formal tool technique to "cut" any hierarchical signal in the design. The "cut-signal" will be considered as primary control point for the formal engines and can freely toggle. To control this signal, we use constraints. Furthermore, we use an additional control signal to control time of fault injection. Below, we discuss permanent faults (stuck-at-0, stuck-at-1), transient, and intermittent faults.

MODELING PERMANENT FAULTS WITH FORMAL

The fault control signal is a primary input to the formal tool and can freely toggle at any time. For controlling fault injection of multiple signals, we use multiple control signals. For simplifying the discussion here, we are considering only the case of controlling one fault signal. We will inject a fault when the control signal, *control_sig*, is high. During this time, the design signal *design_sig*, for which we are injecting fault, will be assign to the value 0 (or 1). When the control signal is low, the design signal will assume the design driven value provided by the signal *ref_design_sig*. The signal *ref_design_sig* is driven by the design copy used as reference and it is the "counter-part" of signal *design_sig*. As the control signal can toggle more than one time, we add a constraint preventing multiple toggles to happen. Note that, (1) the control signal is assumed to be low initially and (2) we are not forcing the control signal to toggle at all and thus leaving the tool deciding to inject faults or not. Figure 5 summarizes the formal modeling of stuck-at 0.




```
assume property  (control_sig   |=> control_sig);
assume property  (control_sig   |=> design_sig == 0);
assume property  (! control_sig |-> ref_design_sig == design_sig);
```

Figure 5 Waveform and assertions to model permanent faults with formal

MODELING TRANSIENT FAULTS WITH FORMAL

The situation is similar to the permanent fault modeling case. However, in this case, we expect to have the control signal toggling two times from low to high and back to low. As before, when the control signal is low there is no fault injection. Instead, fault injection takes place here on time period where the control signal is high. To add more flexibility to the modeling, we have a verification parameter that defines how long the control signal can be high and thus controlling the transient time window.

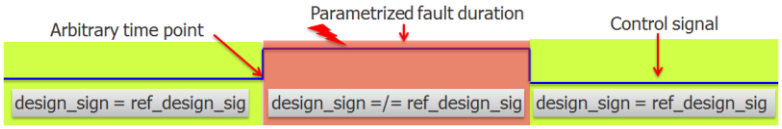


Figure 6 Waveform and assertions to model transient faults with formal

MODELING INTERMITTENT FAULTS WITH FORMAL

Intermittent fault can be seen as a sequence of transient faults. Figure 7 displays a general waveform of this kind of faults. For this kind of fault model, we are using two verification parameters: One parameter is used for the duration of fault injection and the second is used to control the time duration between two consecutive faults.

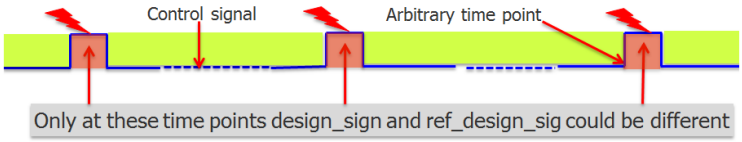


Figure 7 Waveform and assertions to model intermittent faults with formal

V. RESULTS

We implemented the formal fault injection flow using the sequential equivalence checker from the Mentor's Questa Formal tool set. We also used the same DUV used to demonstrate fault simulation in Section IV.B. We run two different safety configurations. In the first one, we specified to have faults occurring only on the clock generator block. Table 2 summarizes the results of running formal fault verification for this configuration. As already explained in Section IV.B, the column "Fault Category" lists the kind of design elements considered as fault. In our case here and as the TMR's used in this design are supposed to detect all faults in the clock generator and propagated only correct values, we have expected results in column "Non-propagatable to Safety Mechanism Outputs". In the column "Non-propagatable to Safety Mechanism Inputs", we see different number of faults that can, in some scenarios, not reach the safety mechanism input interface. This is an important fact and shows that if we use fault simulation with test cases that exactly cover these scenarios then we run into the risk not to correctly verify these faults.

Table 2 **Formal Fault Injection Results –Assume faults in clock generator**

Fault Category	**Formal Fault Injection Results**			
	Number of faults	***Run time***	***Non-propagatable to Safety Mechanism Inputs***	***Non-propagatable to Safety Mechanism Outputs***
All nodes	2648	04h:25m	90.81%	100%
No internal nodes	1428	49m.51s	88.28%	100%
Cell outputs	449	18m:33s	90.64%	100%
Storage elements	57	15m:50s	84.21%	100%

In the second safety configuration, we extend the scope of fault occurrence to be the complete design (including the TMR's). Table 3 summarizes the results of this case. As expected, here we see that faults can be propagated to outside the safety mechanism and thus compromise the design functionality. Figure 8 shows how a fault injected shortly after reset removal reached the safety mechanism input interface after five clock cycles and then reached the safety mechanism output interface.

Table 3 **Formal Fault Injection Results –Assume faults occur everywhere in design**

Fault Category	Formal Fault Injection Results			
	Number of faults	***Run time***	***Non-propagatable to Safety Mechanism Inputs***	***Non-propagatable to Safety Mechanism Outputs***
All nodes	12963	22h:12m	93,55%	92,23%
No internal nodes	7082	11h:17m	94,21%	95,11%
Cell outputs	2206	83m:49s	97,59%	94,36%
Storage elements	267	22m:57s	96,62%	87,64%

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

VI. ISO METRICS

The results above could be used in the calculation of ISO 26262 architectural metrics SPFM and LFM and the probabilistic metric PMHF. Since the design has a safety mechanism in place, by definition no Single Point Faults (SPF) exists. The formal method used above proved that no faults from any of the clock generators could propagate to the outputs. Hence the percentage of residual faults and latent faults in the clock generators themselves is 0. However, faults in the majority voter itself could result in either a residual fault or latent faults. For example, a fault happening in the final output level after the majority vote will propagate and become a residual fault. In a similar way, a fault that sets the majority vote result into constant false positive, would become a latent Multi Point Fault (MPF). The results from our formal analysis could help us show that the percentage of residual faults and latent MPF in the comparator are as specified in the following table. These results could be used to calculate the overall metric values for the TMR and majority voter.

VII. SUMMARY

Safety mechanisms are one of the most critical areas of ISO 26262 compliant automotive designs and their architecture and quality are a key differentiator for various IC providers. This implies that they should be verified as rigorously as possible, and their efficiency in detecting and correcting faults thoroughly and accurately analyzed. While verifying a sample of possible use cases might be sufficient for less critical areas, safety mechanisms require more rigorous means. Formal engines have the ability to fully analyze the entire state spaces, and are therefore a natural fit both for verifying functionality and analyzing efficiency of safety mechanisms. The application of formal to a TMR safety mechanism presented in this paper, is an example of how through verification and relevant metrics could be obtained with relatively little effort and minimal run time. It is generic enough to be adapted for other safety mechanisms, therefore allowing them to achieve the same quality of verification and analysis.

REFERENCES

[1] The ISO26262 Standard. Road vehicles, Parts 1-10, 15 Nov. 2011.

[2] SPI Bridge, OpenCores (www.opencores.org)

[3] D. Brand. “Verification of large synthesized designs.” Proc. ICCAD ’93, pp. 534 -537.

[4] OneSpin Solutions. Fault Propagation Analysis. https://www.onespin.com/products/specialized-apps/fault-propagation-analysis/

[5] Othmane Bahlous and Abdelouahab Ayari. Supplementing Simulation of a Microcontroller Flash Memory Subsystem with Formal Verification. In Proceedings of DVCon 2012, San Jose, CA, USA, February 2012.

[6] Richard Boulton and Mark Handover. Cost Evaluation for Adopting Formal Property Checking: A Detailed Case Study. In Proceedings of DVCon 2009, pages 200–205, San Jose, CA, USA, February 2009

[7] Jörg Grosse, OneSpin Solutions and Mike Bartley TVS. Verifying Functional, Safety and Security Requirements for Standards Compliance. In Proceedings of DVCon Europe 2015, Munich, Germany, November 2015.

[8] Holger Busch, Infineon Technologies. Automated Safety Verification for Automotive Microcontrollers. In Proceedings of DVCon 2016, San Jose, CA, USA, February/March 2016.

[9] Holger Busch, Infineon Technologies. Quantification of Formal Properties for Productive Automotive Microcontrollers. In Proceedings of DVCon 2013, San Jose, CA, USA, February/March 2013.

[10] Holger Busch, Infineon Technologies. An automated Formal Verification Flow for Safety Registers. In Proceedings of DVCon Europe 2015, Munich, Germany, November 2015.

[11] Tun Li, et al. School of Computer, National Universtity of Defense Technology, P.R. China. Application specified soft error failure rate analysis using sequential equivalence checking techniques. In Proceedings of ASP-DAC 2013, Yokohama, Japan, January 2013.