Automated Safety Verification for Automotive Microcontrollers

H. Busch
Infineon Technologies AG
Am Campeon 1-12
85579 Neubiberg, Germany

***Abstract*-Automotive microcontrollers provide new advanced but highly safety-critical features for driver assistance. Therefore they are furnished with various safety mechanisms, and their complete design process follows the ISO 26262 standard. Depending on the targeted safety integrity level, an according high percentage of random faults with the potential to violate safety goals must be provably detectable, and configurable reactions be provided in order to drive the microcontroller system into a safe state. If required by the safety application, vital data needs to be corrected in case of faults. Safety enhancements which are inserted at later design stages must be guaranteed to not corrupt the regular functional behavior of the microcontroller. This paper discusses an automated formal safety verification methodology which uses formal property checking and includes fault injection for verifying the correct installation of hardware safety mechanisms, and summarizes applications to components of a new Infineon microcontroller product family.**

I. INTRODUCTION

This paper is organized as follows. In this section, the motivation of this work is discussed, and background information given. In Section II, a few hardware safety mechanisms comprised in recent Infineon Microcontrollers and which are appropriate for formal verification are summarized. Section III details our corresponding formal safety verification approach. Section IV describes how we ensure that mission function is preserved when safety measures are added later. Section V discusses application and results, Section VI draws conclusions.

A. Impact of ISO 26262

The ISO 26262 standard is focused on functional safety of safety-related electric and electronic systems in mass produced passenger cars up to 3.5 tons against potential hazards due to malfunction in electronic and electronic systems and their interaction. Since its introduction in January 2012, the standard obliges car manufacturers and their suppliers including chip suppliers to develop all new products according to safety requirements in so-called work products of the standard. The standard has a significant impact on processes, documentation, and technologies deployed during the development of safety-relevant systems, ranging over the complete life-cycle from initial development phases to decommissioning of the final product.

The standard specifies 4 automotive safety integrity levels (risk levels) ASIL A-D which are derived from exposure (probability of exposure of hazardous event), severity (of resulting harm), and controllability (chance for driver to avoid harm), and which classify the requirements of an item to avoid an unreasonable risk. For each risk level, the standard defines random hardware failure target values ranging from $< 10^{-8}$/hour = 10 FIT (failure in time) for ASIL D to $< 10^{-6}$/hour = 1000 FIT (failure in time).

For certification, the diagnostic coverage needs to be assessed in order to show that the percentage of the failures of a safety-related element that are detected or controlled by implemented safety mechanisms is sufficiently high according to the ASIL classification. For ASIL D, the percentage of all single-point and residual faults with the potential to violate the safety goal must be <= 1%, i.e. 99% diagnostic coverage is required.

Diagnostic coverage is typically determined by fault-simulation, which uses fault-injection, and reruns test-cases many times with different activated faults. This is computationally very expensive, even though modern fault-simulators include algorithms for saving simulations for related faults and using statistical samples rather than full enumeration. An interesting question is what formal verification can contribute to diagnostic coverage computation, since fault injection can also be done in formal models. While formal property checkers today can't directly generate statistical detection rates like fault simulation, they can prove 100% diagnostic coverage for elementary parts fully protected by specific safety mechanisms.

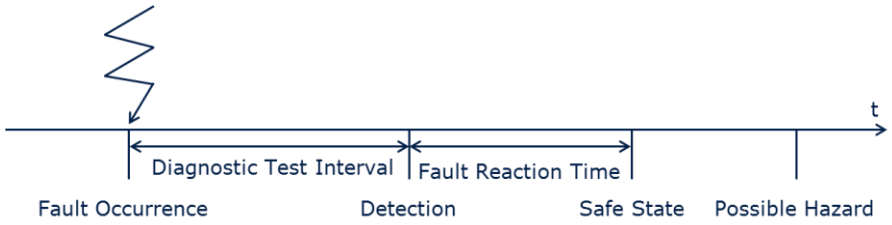

Figure 1. Definition of Time Intervals in ISO 26262.

B. Formal Safety Verification

Apart from verifying the effectiveness of safety mechanisms, formal safety verification can ensure that the maximum diagnostic time interval, i.e. the period of time between fault occurrence and fault detection, and fault reaction time, i.e. the time after fault detection for driving the controller in a safe state, do not exceed the maximum values required in the safety concept. In model-checking no time units like milliseconds are applicable, but instead clock cycles, physical latencies can only be computed indirectly from given clock frequencies outside of the formal tool.

Due to FMEDA results (Failure Modes and Effects Diagnostic Analysis), which comprise analyses of the effects of random hardware faults and estimates of failure rates and the probabilities and rates of the violation of safety goals, it may turn out that safety mechanisms have to be added or strengthened, after the regular function has already been implemented and verified. In turn, it may also happen that a safety measure is replaced, reduced, or removed, if the overhead in terms of area and power consumption is not justified by the gained overall safety level, or if the application of a specific derivative product of a microcontroller family has less strict safety requirements. Subsequent safety augmentations or modifications must not corrupt the soundness of the design implementation and previous verification results.

Any functional safety modifications can be verified by constrained sequential equivalence checking. However, rather than using a separate tool, we augment our existing property-checking set-up and use an own dedicated methodology for verifying the preservation of previously verified function.

C. Tool Environment

Our formal safety verification methodology is built on top of (but not restricted to) the property-checker from Onespin Solutions, which is a bounded model-checker. We regularly use it for IP-level functional verification of system control modules of automotive microcontroller systems. Onespin's tool environment offers a rich set of functions which allow us to create own tailored proof automation procedures for specific purposes like safety verification.

The property checker 360 DV-Verify has been applied for many years to numerous Infineon module designs. For most of the formally verified modules no additional simulation-based verification was performed, while in turn, insufficient functional coverage of simulation has been complemented by formal property checking of specific aspects.

Onespin's property checker has front-ends for VHDL, Verilog, SystemVerilog and since very recently SystemC. It understands different property languages.

- SVA
- PSL
- ITL (InTerval Logic, Onespin's proprietary property language) properties with state expressions in VHDL or Verilog syntax

Property sets written in ITL may be combined with sets of SVA and PSL assertions, and environment constraints be mixed as well. Several different proof engines are available which are optimized for different purposes. For instance, one engine is specialized on generating counterexamples (of failed proofs) or witnesses (of successful proofs) reachable from reset. Another engine proves properties from any start state, however may return unreachable signal traces. When a property check is invoked, the user can specify a set of proof engines to be tried in parallel or sequentially.

User-defined reset sequences, periodic or free-running clocking schemes can be specified for any number of clocks, and attributes for design inputs can be entered. The type domains of all signals can be specified in compile options, so that checks can be restricted to 2-valued logic or include Z- and X-values, e.g. in order to investigate X-propagation.

Like other commercial property checkers, Onespin offers linting and a formal consistency checker which identifies dead code, in addition redundant code, sticky signals, checks of FSMs, in-code assertions extracted from the design, and several other automatically generated assertions. Several standard "apps" mainly address verification engineers with less deep background in formal verification. They just need to run predefined automated property generators and proof scripts for standard checks of X-propagation, register-function, connectivity, and clock-domain-crossings.

Onespin's extensive debugging features are very helpful for general-purpose property checking. A root cause analyzer allows the GUI user to follow fan-ins and fan-outs over different clock cycles. Properties and RTL source code are annotated with signal values from counter- or witness-traces, and code regions active in a clock cycle selected in the waveform viewer are highlighted.

The 360 DV-Verify environment comprises a formal completeness checker which applies the strongest possible criteria for gap-free verification, such as full case split over all input scenarios and seamless output determination. These criteria, if fulfilled, altogether establish sequential input-output equivalence of properties and design. This formal completeness notion is not compatible with coverage metrics used in simulation environments, and a strictly structured suite of formal operation properties has to be written in order to run formal completeness checking, which is not always feasible within tight project schedules. Another feature called Quantify was therefore added which performs coverage analyses closer to the classical notions used in simulators. When doing functional verification, this "metric-driven verification" feature, which is based on mutation coverage and requires no specifically structured property sets or user interaction, is very useful to identify areas of RTL not yet sufficiently covered by properties.

As the Onespin GUI comprises a TCL-shell and a rich library of useful TCL utilities allowing evaluation and control of internal data such as filtered signal lists, bit-precise fan-ins, property and constraint lists, or current proof status of properties, users can well add own TCL functions for their purposes.

Onespin also makes model mutation available to users by allowing any arbitrary port and internal signal at any level of hierarchy to be cut by way of a compilation option into a fan-in part with the original signal's fan-in function, which becomes a new external output of the module, and a fan-out part, which is replaced by an external input which drives the signal's fan-out. In added property assumptions, any behavior of the artificial inputs can be assumed, including the normal signal behavior if input and output parts are just connected. For formal fault-injection, this cut-signal feature is crucial by allowing fault models to be specified in a very flexible way.

D. Summary of Our Formal Safety Verification Methodology

Our methodology comprises automatic proof functions for verifying safety mechanisms which include error detection, error correction, and self-test features controlled by software which generate expected alarms distinguishable from spurious alarms caused by real faults. Self-test features are verifiable without fault-injection by way of properties which check for the occurrence of expected alarms when a self-test is activated and for the absence of alarms during regular operation. For this purpose, pre-defined and highly re-usable generic properties are fed with macros automatically generated from the design.

By way of another property set based on fault injection, the proper handling of spontaneous faults is verified. Again, these properties are filled with generated macros.

In order to prove that the installation of a hardware safety measure does not affect the regular function, another approach is presented in this paper which is based on formal equivalence properties proven on an automatically generated wrapper architecture with two different instances of original and modified design.

E. Related Work

In [3], we gave a first early outline of our formal safety verification approaches, which have since then evolved considerably and been applied on more modules. In [3], we presented our combination of a test-bench qualification methodology based on Certitude from SYNOPSYS with Onespin's formal property checker. For running property regressions after safety augmentations or modifications of designs fully functionally verified by formal properties, we use this methodology on a regular basis. An introduction to concepts of formal property completeness and coverage was given in an Onespin tutorial at DVCON [2]. In [5], we elaborated on an automated formal verification methodology for safety-critical special function registers based on XML-specifications of functional and safety-related aspects of software-accessible registers. As registers hidden to software may also contribute to vital function, the work presented here addresses the safeguarding of any internal register. In [6], a tutorial was given which particularly addresses tool support for ISO 26262 compliance. Due to the need to provide evidence for ISO compliance, such tools which support requirement traceability are being used especially in the area of functional safety for safeguarding and confirming the fulfillment of functional safety requirements.

II. HARDWARE SAFETY MECHANISMS

In this section we give an overview of the basic register monitoring architecture implemented for safety-critical registers.

We replace ordinary registers with ones embedding redundancy and self-test logic (SFF – safety flip-flops), which may be single bits or registers of any width.

In order to achieve short diagnostic time intervals [1], we provide alarm reduction logic which combines alarms from different safety registers into compound alarms at different levels of hierarchy.

A test-controller (SFFTC) combines the received alarms, performs some synchronization into a target clock domain, if required, and sends a compound alarm pulse to a safety management unit (SMU), which contains software-configurable logic for triggering appropriate actions within a short fault reaction time in order to drive the microcontroller system into a safe state.

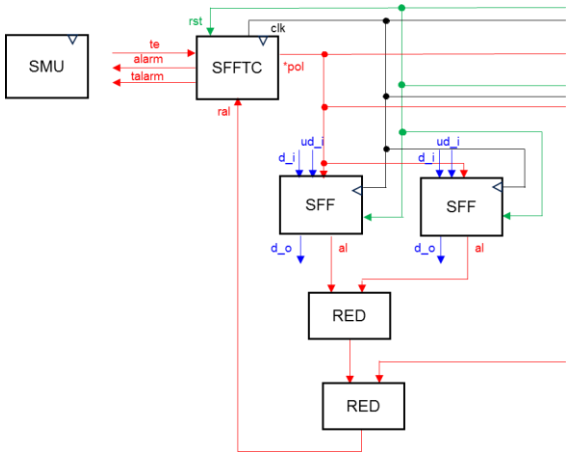

Figure 2. Safeguarding Structure

From time to time, the SMU can enable self-testing (te), in order to check whether the safety mechanism is still fully functional. While the self-testing is enabled, the test-controller will issue special control signals which force the register monitoring logic to generate expected alarms (alarm). If the expected alarms do not fully match with the actually received alarms, the test controller will actiate a distinct test-alarm (talarm) which indicates to the SMU that the register monitoring has some fault. Safety software might then decide to reset a part of the system and repeat the self-testing in order to distinguish an intermediate from a permanent fault.

The registers that are built from safety flip-flops (SFF) contain different kinds of redundancy, some local fault detection logic, i.e. some kind of comparators, self-test-logic triggered by test-inputs which will manipulate register contents in a well-defined way, and in some variants also fault correction logic. Accordingly, in some embodiments there are separate alarms for correctable and uncorrectable errors.

Redundancy may range from a single parity bit to duplication (DMR – double modular redundancy), triplication (TMR – triple modular redundancy), ECC, etc.

All these approaches have in common that at least any single bit error is detected, but only some will restore register contents in case of correctable errors, e.g. TMR (by majority vote), and ECC versions with correction capability.

The decision for a specific safeguarding approach is not straightforward. A TMR solution with triplication of register bits tends to require more area than an ECC solution where much less redundant register bits are needed. However, it is not that simple. For instance, ECC encoding and decoding logic requires more combinatorial gates, and the combinatorial propagation delays can be a problem in higher clock frequency domains. So the optimal solution depends on various characteristics of the different basic library cells like area and capacitances, on the optimization capabilities of design compilers, placement constraints, and the frequency ranges for the different clock-domains of safety-critical components. Moreover, the assessment of the really required safety measures for individual elements depends on difficult cause-effect analyses at different levels of abstraction. Before the lay-out is finished, actual area overhead and propagation delays can just be estimated.

From this follows that the selection of a specific hardening approach for an individual safety-critical function may not be stable over all product versions and derivatives, and that there is a need to verify that safety modifications do not affect any mission function.

III. FUNCTIONAL VERIFICATION OF HARDWARE SAFETY MECHANISMS

The verification of the effectiveness of the register monitoring is done in different set-ups, without and with fault injection.

A. Safety Verification without Fault-Injection

Without fault-injection, only a part of the safety mechanism can be verified. Alarms can only be triggered in self-testing mode, and there should be no unexpected alarm, provided the safety mechanism has been properly installed.

Essentially three corresponding proof goals are to be proven. Goal (1) states that when the test mode is disabled for at least a self-test interval Tst, no alarm is asserted at the end of this time interval plus an additional diagnostic time interval Tdti. In other words, with self-test having been disabled for some time, no alarm is asserted thereafter:

$$\forall_{t1:\, t\le t1\le t+Tst}\ te(t1) = 0 \Rightarrow alarm(t+Tst+Tdti) = 0 \qquad (1)$$

The second goal (2) specifies that when the self-testing is enabled, an alarm is flagged within a time interval of self-test duration Tst after a latency of Tdti cycles:

$$te(t) = 1 \wedge \forall_{t2:\, t\le t2\le t+Tst+Tdi}\ reset(t2) = 0 \Rightarrow \exists_{t3:\, t\le t3\le t+Tst+Tdi}\ alarm(t3) = 1 \qquad (2)$$

Goal (3) excludes an unexpected alarm (without fault injection)

$$true \Rightarrow talarm(t) = 0 \qquad (3)$$

In all these formulae, t is implicitly all-quantified. In (2), it is necessary to exclude reset, which could otherwise interrupt the self-test. In the other 2 proof goals, a reset would not cause a violation of the conclusions, therefore it does not matter whether a reset occurs or not, since in both cases no alarm or test alarm would be activated.

Although looking simple, these properties can have a huge fan-in, as several hundreds of registers with completely different regular function may be safeguarded by a single test-controller with a corresponding vast state space to be traversed. It is not the self-testing that potentially causes complexity but a possibly inconsistent state which would be visible in an active test-alarm, which could occur in any safeguarded register bit with possibly very high sequential depth. Thus we follow an inductive approach by splitting goal (3) into base and step case.

$$reset(t) = 1 \Rightarrow \forall_{t1:\, t\le t1\le t+Tst}\ talarm(t1) = 0 \qquad (3a)$$

$$\forall_{t1:\, t\le t1\le t+Tst}\ talarm(t1) = 0 \Rightarrow talarm(t+Tst+1) = 0 \qquad (3b)$$

The reason for this splitting is that a less expensive proof engine starting from an arbitrary state can be used this way.

Note that it is necessary here to prove the base case for several cycles, and make the corresponding assumption in the step case. Goals (1) is split in the following way:

$$reset(t) = 1 \wedge \forall_{t1:\, t\le t1\le t+Tst}\ te(t1) = 0 \Rightarrow \forall_{t2:\, t\le t2\le t+Tst+Tdti}\ alarm(t2) = 0 \qquad (1a)$$

$$\forall_{t1:\, t-Tdti\le t1\le t+Tst-Tdi}\ talarm(t1) = 0 \wedge \forall_{t2:\, t\le t2\le t+Tst}\ te(t2) = 0 \Rightarrow \forall_{t3:\, t\le t3\le t+Tst+Tdti}\ alarm(t3) = 0 \qquad (1b)$$

By excluding an active test alarm for several cycles, it is ensured that no inconsistent previous state is taken, and the result can be proven also for (1b) without reset assumption

The proof of an alarm in test mode is split as follows:

$$reset(t-1) = 1 \wedge \forall_{t1:\, t\le t1}\ reset(t1) = 0 \wedge \exists_{t2:\, t\le t2\le t+x}\ te(t2) = 1 \Rightarrow \exists_{t3:\, t+Tdti\le t3\le t+x+Tst+Tdti}\ alarm(t3) = 1 \qquad (2a)$$

$$\forall_{t1:\, t\le t1\le t+Tst+Tdti}\ reset(t1) = 0 \wedge \forall_{t2:\, t-Tdti\le t2\le t+Tst-Tdti}\ talarm(t2) = 0 \wedge te(t) = 1$$
$$\Rightarrow \exists_{t3:\, t+Tdti\le t3\le t+Tst+Tdti}\ alarm(t3) = 1 \qquad (2b)$$

A reset is just excluded for the whole examination window. These properties ensure that all self-test and fault detection logic included in the alarm reduction is properly installed and that no false alarm or test alarm is generated by the circuit. However, they alone do not guarantee that each safety register is actually included in the alarm reduction.

B. Safety Verification with Fault-Injection

Whether a local alarm signal is actually captured in the reduced alarm can only be checked by injecting faults into safety registers. The big advantage of formal verification in comparison to simulation is that fault injection can be done exhaustively and that any corner case in which an alarm would not be propagated to the SMU will be detected.

As we have designed all safety registers in such a way that they can be easily filtered from the list of all signals, after the design has been elaborated by the property checker's HDL front-end, we have been able to develop functions for generating compile options in order to cut exactly the signals related to safety registers. Other functions generate macros which express whether individual productive or redundant register bits are connected or put into some fault state according to the fault model to be checked. By combining all productive and all redundant bits into vectors, generating additional fault modelling vectors, we can assume any combination of errors and check whether they are detected.

In the figure below, we show the cut model for double- and triple-modular-redundant safety registers, the latter version comprising correction of single bit errors by majority decision.

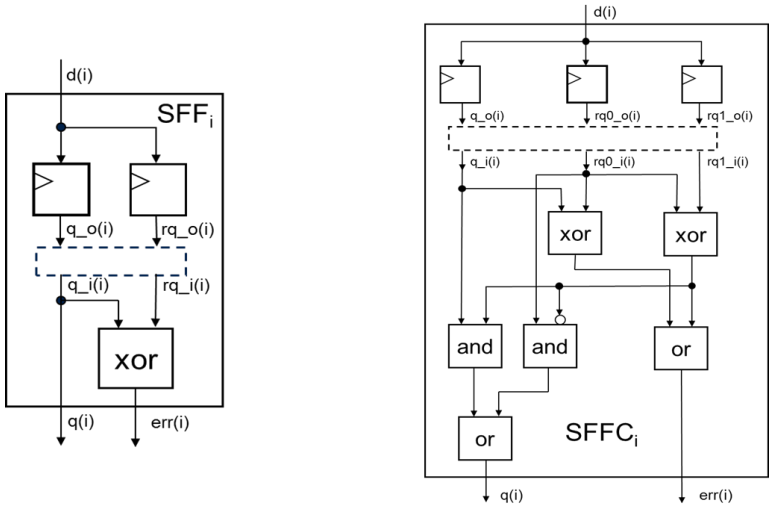

Figure 3. Safety Flip-Flops for Detection (DMR) and Correction (TMR).

After combining all these bits for a potentially great number of SFF into automatically generated vectors, we can specify by a Boolean function that n bits are flipped:

be(n) := Σi (q_o || rq_o)(i) xor (q_i || rq_i)(i) = n

Thus be(0) expresses that all cut signals are connected, be(1) that 1 bit is flipped, be(n) that n bits are flipped.

Correspondingly for triple modular redundancy, the two redundant bit vectors are concatenated to the vector with the productive bits.

be(n) := Σi (q_o || rq0_o || rq1_o))(i) xor (q_i || rq0_i || rq1_i)(i) = n

The XOR function generates difference vectors, of which the number of 1-positions indicates how many errors have been injected. It would be possible to iterate explicitly over all positions of the difference vectors, and check individually whether each time an alarm is raised. Instead it is much more efficient to do all these check at once by just specifying by be(n) that n errors occur in whatever combination.

It is obvious that properties (1-3) are still provable if just the assumption

$\forall_{t1:\ t\le t1\le t+Tst+Tdti}$ be(n)(t1) = 0 is added.

If we assume a single bit error, we can now prove that alarm and test alarm are flagged

$\forall_{t1:\ t1<t}$ be(0)(t1) $\wedge$ talarm (t1) = 0 $\wedge$ be(1)(t)

$$\Rightarrow \exists_{t2:\ t\le t2\le t+Tdti} \text{alarm}(t2) = 1 \wedge \exists_{t3:\ t\le t3\le t+Tdti} \text{talarm}(t3) = 1 \qquad (4)$$

In the same way we can assume any numbers of faults. However, we have to make sure in case of double modular redundancy that not only pairs of productive and corresponding redundant bits are flipped, as such errors are not detected. This is achieved by forming difference vectors of the difference vectors of all redundant bits and all productive bits. We can then actually prove that if this second order difference vector has at least one 1-position, an alarm and a test alarm is raised.

Correspondingly we generate for TMR safeguarding higher-order difference vectors which ensure that not all triples are either all flipped or not flipped.

The question of course is how likely it is that an alpha particle hits exactly just pairs of DMR or triples of TMR flip-flops so that all are consistently flipped. By some measures this is made very unlikely, like for instance inverted implementation of one redundant bit and different test logic causing physical separation in the lay-out.

While we have concentrated so far on the alarm generation, register safety mechanisms with correction also need to be checked for recovery of distorted data.

As we use uniform correction mechanisms, we are able to automatically filter out the signals which represent the register values behind the correction logic and which are functionally being used. Again these register read signals are combined into vectors which can be related to the difference vectors. For this purpose, we introduce another Boolean function for multiple bit errors in triples:

mbe(n,m) := $(\Sigma i\ ((q\_o \text{ xor } q\_i)(i) + (rq0\_o \text{ xor } rq0\_i)(i) + (rq1\_o \text{ xor } rq1\_i)(i) = m)) = n$

Here we first count the number of errors in each bit of each triple and check whether this bit sum equals to parameter m. If so, we add a 1 to the overall sum of all positions which have m errors. If this overall sum is n, then the Boolean function returns true, else false. In the sum expression we neglect type conversions between Boolean, bit and natural numbers. With this definition we can write a property with an assumption that n triples have single bit errors, and no double or triple bit errors occur.

$$\forall_{t1:\, t \le t1 < t}\ be(0)(t1) \wedge talarm\ (t1) = 0 \wedge mbe(n,1)(t) \wedge mbe(0,2)(t) \wedge mbe(0,3)(t) \Rightarrow \forall_{t1:\, t \le t1 \le t}\ q = q\_i \qquad (5)$$

Under this assumption, it can be proven that the corrected read vector is identical to the real functional register value before fault injection.

Alternatively to DMR/TMR safeguarding, registers can be safeguarded by ECC (Error correcting code). Typically ECC-codes are used for SRAMs with safety-critical data with a variety of data widths. Nevertheless, we also protect a few safety registers by ECC.

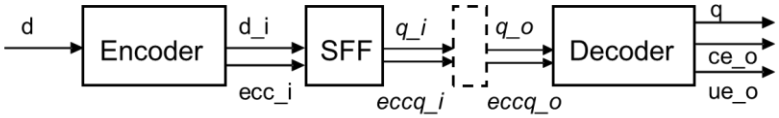

Figure 4. ECC Protection of Registers

When using ECC protection for registers, we can install in principle the same alarm reduction architecture as shown before according to Figure 2. Here correctable and uncorrectable errors are distinguishable, if an appropriate ECC approach is selected. SECDED (single error correction, double error detection) is well established and mathematically understood to be capable of correcting 100% of all single-bit errors in a code word, and of detecting 100% of any combination of 2-bit errors of the code word, which is advantageous when it comes to providing evidence for diagnostic coverage of the installed safety mechanisms. By applying an appropriate ECC approach, we can send separate reduced alarms for correctable and uncorrectable errors to the SMU. Generally, less redundant register bits are required than for duplication or even triplication. For instance, if 32 register bits or the bits of a 32-bit SRAM data word are to be protected by SECDED, an ECC width of 7 is sufficient. However, more combinatorial encoding and decoding logic is obtained, which adds propagation delays to be carefully taken into account especially for higher frequencies. Depending on the available ECC codes, differently sized safety registers can be protected. Safety registers can also be concatenated in order to protect them by common ECC-logic.

For the safety verification of the ECC logic, we specify a similar Boolean function to express how many bit errors have occurred:

be(n) := $\Sigma i\ (q\_i \,||\, eccq\_i)(i)\ \text{xor}\ (q\_o \,||\, eccq\_o)(i) = n$

With this definition a property for correctable errors is specified as follows, here assuming an ECC with single bit error correction and distinct correctable-error flags alarm_ce and talarm_ce.

$$\forall_{t1:\, t \le t1 < t}\ be(0)(t1) \wedge talarm\_ue(t1) = 0 \wedge talarm\_ce(t1) = 0 \wedge be(1)(t)$$
$$\Rightarrow \exists_{\,t2:\, t \le t2 \le t+Tdti}\ alarm\_ce\ (t2) = 1 \wedge \exists_{\,t3:\, t \le t3 \le t+Tdti}\ talarm\_ce\ (t3) = 1 \wedge \forall_{t4:\, t \le t4 \le t+Tdti}\ q(t4) = q\_i\ (t4) \qquad (6)$$

If the ECC detects double bit errors, it cannot be decided whether ECC or data bits are affected. Thus it can just be proven that a reduced uncorrectable error is flagged to the SMU:

$$\forall_{t1:\, t\le t1<t}\ be(0)(t1) \wedge talarm\_ue(t1)=0 \wedge\ talarm\_ce(t1)=0 \wedge be(2)(t)$$
$$\Rightarrow \exists_{t2:\, t\le t2\le t+Tdti}\ alarm\_ue\ (t2) = 1 \wedge \exists_{t3:\, t\le t3\le t+Tdti}\ talarm\_ue\ (t3) = 1 \qquad (7)$$

For single faults all these safeguarding approaches and with detection and optionally with correction allow automated proofs of 100% diagnostic coverage for single faults in any of the safeguarded register bits. However, while statistical evaluation of simulations of multiple bit errors (>2) with a detection rate of less than 100% is possible, formal verification cannot directly contribute such diagnostic coverage figures. For instance, typical SECDED algorithms achieve up to70% detection of 3-bit errors and their correct classification as uncorrectable errors. Here it would be e.g. too inefficient to generate and iteratively exclude the counterexamples for 30% of all possible faults until a property proving correct uncorrectable-error-alarms for the remaining 70% could be proven.

IV. FUNCTION PRESERVATION OF SAFETY MODIFICATIONS

Whenever safety mechanisms are added or modified, regular function is potentially affected by newly introduced corner-cases. It could for instance turn out that instead of mere detection, correction is mandated by safety assessors for particularly critical register bits. However, if correction or self-test logic itself is flawed even for just few potential combinations, systematic errors can be introduced which outweigh the intended gain of safety. Therefore it is important that such safety logic modifications are very thoroughly verified.

If a complete property set is available, it is possible to just rerun the property checking regression on the safety-enhanced design. The completeness of property sets can be formally proven by applying Onespin's formal completeness checker. A less strict assessment of property sets is feasible with Onespin's formal coverage checker Quantify, or by way of the fault instrumentation tool Certitude. Hence, these completeness-checked property sets allow unexpected functional side-effects of late safety augmentations to be excluded with high reliability. If a module is verified with a simulation test-bench, the Certitude-based approach is applicable to modules not having been formally verified.

However, it may be desirable to split the responsibilities for the functional verification of designs and the safeguarding of their safety enhancement to independent specialized verification groups. An alternative approach already applicable when a complete formal verification has not yet been carried out is sequential equivalence checking with fault injection. We have devised a dedicated procedure which comprises a pre-analysis step, generation of a wrapper-architecture with two design versions, and the generation of invariant properties for equivalence checking.

The pre-analysis step checks which RTL source files have changed, using standard text comparison functions. The lowest instance in the component instance hierarchy which contains all modified components is chosen as top-level for the equivalence check. We may also decide to black-box unchanged components of this sub-architecture.

In the next step, a wrapper architecture is generated which includes the chosen sub-architectures of original and modified design. The wrapper entity gets the union of the inputs of both sub-architectures as input ports, and the two sets of outputs of both sub-architectures as output ports. If black-boxing is applied, the inputs of the black-boxed components are treated as outputs of the corresponding sub-architecture, and the outputs of the black-boxed components as inputs of the sub-architecture. In the wrapper architecture, the wrapper inputs inherited from common inputs of the sub-architectures are connected to both sub-architectures. The remaining inputs are wired through just to the corresponding inputs of one of the two sub-architectures.

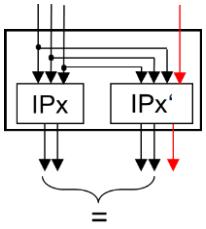

Figure 5. Wrapper Architecture with Original and Safety-Augmented Module.

B. *Equivalence Checking without Fault-Injection*

The equivalence check is first set-up without fault injection by the following proof goal which states that all outputs of previous and safety-enhanced architectures are equivalent at any time:

$$\text{true} \Rightarrow \text{ipx\_o}\ (t+1) = \text{ipx'\_o}(t+1) \tag{8}$$

As this proof goal is in general too complex if the reachable state space from a separately specified reset sequence is traversed for a longer proof radius, it is replaced by an inductive approach with separate goals for the base case (8a) and the step case (8b) and which include the internal states of original and augmented design.

$$\text{reset}(t) = 1 \Rightarrow \text{ipx\_o}(t) = \text{ipx'\_o}(t) \wedge q(t) = q'(t) \tag{8a}$$

$$\text{ipx\_o}(t) = \text{ipx'\_o}(t) \wedge q(t) = q'(t) \Rightarrow \text{ipx\_o}\ (t+1) = \text{ipx'\_o}(t+1) \wedge q(t+1) = q'(t+1) \tag{8b}$$

It is understood that extra outputs due to the safety augmentation are not considered. In addition to the outputs ipx_o, ipx'_o, register read signals q, q' are included in the invariant proofs. Hence, even if the internal implementation of the registers with different or no redundancy and self-test logic differs in both architectures, the register values fed into the fan-out logic must be identical.

All required signals are filtered automatically so that the corresponding properties and checks can be generated automatically.

A. *Equivalence Checking with Fault-Injection*

There are several use-cases of equivalence checking with fault injection:

1. Comparison of 1st original unprotected architecture with a 2nd safety-enhanced architecture with correctable faults being injected in the 2nd.
2. Comparison of 1st safety architecture with a 2nd modified safety architecture with correctable faults being injected in both architectures. The injected faults in the architectures can be completely independent; they must only be correctable.
3. Comparison of a safety-architecture with itself with correctable faults being injected in only one instance.

An example for Use-case 2 is the replacement of a TMR- with an ECC- or a parity-bit-safeguarding approach. If the correction really works, which should have already been proven on the safety-architectures according to Section III, all functional outputs should be provable to be equivalent if the safety modifications are sound.

For equivalence checking, fault injection can be performed in two different ways:

1. Using be / mbe predicates as shown above to express that only correctable errors occur.
2. Assuming that correctable-error alarm occurs.

Since it has been already proven that 1.) implies 2), and that not 1.) implies not 2.), both versions can be used, but 2.) tends to add more complexity.

For uncorrectable faults the safety mechanism does not claim the function to be preserved, therefore normally no equivalence checks are done with uncorrectable-fault injection. Only in the special case that any effect of a specific uncorrectable fault on a subset of the outputs should be excluded, it makes sense to configure and run a corresponding equivalence check.

The specific equivalence checking approach does not aim at replacing full-fledged sequential equivalence checkers. Nevertheless, for checking the implementation of local safety enhancements, it is advantageous to have maximum flexibility with the injection of faults. Moreover, we can use our familiar formal verification environment with all debugging support from general-purpose formal property checking. Due to the hierarchical alarm reduction architecture the complexity can be controlled by first restricting the equivalence check to lower levels of the instance hierarchy down to the sub-components in which the safeguarding logic is installed.

V. EXPERIENCE AND RESULTS

We have applied the suite of proof automation routines to different system control modules, including memory test and control units, safety management unit, system register unit, parts of clock control and reset control units, and others, the biggest ones with roughly 40 k lines of VHDL code and 1500 protected safety register bits.

The runtimes of all properties summarized in this paper are acceptable: The reset base case proofs range from few seconds to 21 minutes, the step cases from few minutes to about 2 hours. It should be emphasized that in these proofs a vast number of arbitrary fault combinations in thousands of register bits is covered in just one property, by not specifying the fault locations and numbers of faults. In fault-simulation it would be impossible to achieve comparable coverage of fault scenarios in finite time.

In Table 1, proof times for a bigger system control module are given.

TABLE I
PROOF RESULTS FOR TYPICAL SYSTEM CONTROL MODULE

Property	Proof Status	Proof Time (hh:mm:ss)
sff_0_be_alarm_rst	hold	00:01:22
sff_0_be_alarm_step	hold	00:02:33
sff_0_be_no_alarm_rst	hold	00:13:44
sff_0_be_no_alarm_step	hold	02:24:11
sff_0_be_no_talarm_rst	hold	00:21:01
sff_0_be_no_talarm_step	hold	00:01:57
sff_0_be_ok_rst	hold	00:00:04
sff_0_be_ok_step	hold	00:00:26
sff_1_be_alarm_rst	hold	00:00:19
sff_1_be_alarm_step	hold	00:01:45
sff_1_be_talarm_rst	hold	00:00:17
sff_1_be_talarm_step	hold	00:01:17
sff_any_be_alarm_rst	hold	00:00:07
sff_any_be_alarm_step	hold	00:04:11
sff_any_be_talarm_rst	hold	00:00:44
sff_any_be_talarm_step	hold	00:01:27

Each property corresponds to one simultaneous proof for all safety register bits of the module. Properties with the name sff_0_* include the assumption that no error is active, with the name sff_1_* that one arbitrary single bit is erroneous, and the group of properties with sff_any_* even assume any number and combination of bits to be simultaneously flipped. In fact, the sff_any_* properties subsume the sff_1_* properties. Thus it is at first sight surprising that the times for sff_any_* are not noticeably higher.

Proofs of ECC logic depend on the protected code-word lengths. Here the longest property which proofs safe detection of arbitrary combinations of 3 bit errors in a 256-bit data word with 10 ECC bits takes 01:44:49 hours.

As the proofs are simultaneously run in an LSF compute farm with thousands of differently sized and dynamically loaded hosts, the proof times of one property can vary quite a lot depending on the currently available resources. Thus the only reliable message to be inferred from the proof times given in Table I is that compound formal safety verification is very efficient, but the results yielded under real project circumstances cannot directly serve as benchmarks, which is not the purpose of our productive verification work.

The fault injection properties allow us to fill all coverage gaps of function related to the generation of the test alarms. This functionality can simply not be reached without fault injection. Properties with fault-injection but 0-fault assumption run about 30% longer then corresponding properties with regular model without fault-instrumentation.

No complexity issues arose for the property-based equivalence checks since we were able to apply them locally to sub-components whenever new register safeguarding was added.

The verification strongly profited from the uniform safeguarding approach implemented in the verified designs, which greatly eased automatic procedures for extracting key information from the design. In turn, the safety augmentation with encapsulated safety logic allows design engineers to augment their designs safely and in short time. Thanks to the proof routines performing automatic macro generation and using pre-defined safety properties, a few errors were immediately found which concerned missing inclusion of individual local alarms in alarm reduction, and inconsistent clock and reset domains of self-test controllers and connected safety register bits. Additional complexity was caused by the installation of clock-gating for power saving, which required careful analysis of the clock gating conditions during self-test activation and alarm processing.

VI. CONCLUSIONS

Generally, for ISO 26262 certification it is very advantageous to give evidence that a safety mechanism has been exhaustively verified. This is a strong motivation for applying formal verification especially to safety-critical μC components. A well-defined uniform safety mechanism saves a huge amount of design and verification effort. Moreover, once this safety mechanism is certified, the certification of re-used instances is much easier, and ideally can replace repeated safety verification by a rationale.

In this presentation we have shown that formal property-based safety verification has a lot of advantages. In particular, it

- is able to confirm 100% diagnostic coverage of safety mechanisms, enabled by very flexible fault injection capabilities of a formal tool (by design mutation, in this case in the elaborated model) in comparison to forcing of waveform values as typically applied in dynamic simulation.
- is extremely efficient by covering huge sets of faults simultaneously. This high performance even for very long concatenated register vectors is enabled by fan-in analyses.
- allows late safety- or functional modifications to be taken without risk of introducing functional flaws.
- exploits a particularly verification-friendly uniform design-approach which was developed in parallel to the formal verification infrastructure. This uniformity has a positive impact on the design. Due to the encapsulation of safety function, design engineers do not have to care about the internal implementation of the safety mechanism and thus save a lot of effort. In turn, by observing simple naming conventions for signals and their duplicates, design engineers enable automatic filtering functions to be used in generators of macros inserted in pre-defined properties and in proof automation functions.

Our specific equivalence checking approach for safety-augmented designs does not aim at replacing full-fledged sequential equivalence checkers. Nevertheless, it provides

- high computational efficiency due to the flexible hierarchization and localization done by the wrapper generator and by restricting the equivalence check to lower levels of the instance hierarchy down to the sub-components in which the safeguarding logic is installed.
- highest reliability independently of the completeness of property sets, which extends the scope of application to modules not generally formally verified.
- extensive debugging support with all features provided by the property checking environment, which are well known from general-purpose functional property checking.
- maximal flexibility for the injection of faults which can hardly be provided by standard equivalence checkers.

Given the tight schedule and cost constraints of industrial microcontroller development, we are neither able to devise completely new formal verification technology on our own, nor to experiment to a larger extent with unfamiliar verification tools and integrate these in our product development flow, unless we would gain such substantial advantages that the costs and additional efforts for familiarization, integration and adaptation, licensing, etc. would be justified. Instead, we develop own proof automation scripting in our regularly used formal verification flow. As this tool environment offers on the one hand very efficient internal proof algorithms combined with rich and convenient user features, as to be expected from a commercial end-user product, and on the other hand is sufficiently open to allow us to use a comprehensive set of pre-defined utility functions in our own scripts, this setting gives us sufficient flexibility to handle our specific verification problems with tailored solutions developed in parallel to and directly driven by productive design and formal verification work.

While it is important to detect deficiencies of safety mechanisms at early design stages, where design corrections are much cheaper than after tape-out of the product, future work will also address the gate-level by formal safety verification as recommended by the standard in order to make sure that the synthesis has not removed redundant logic. For this purpose we expect to re-use safety properties proven at RTL-level and proof automation functions with just few adaptations at gate level.

ACKNOWLEDGMENT

The work described in this paper is sponsored by the European ARTEMIS funding project EMC^2.

REFERENCES

[1] ISO 26262 Std. Road vehicles, Parts. 1-10, 15 Nov. 2011.
[2] M. Siegel, “Verification Coverage and Productivity Through Formal Operation- and Transaction-Level Verification Using SVA,” Verification Tutorial at DVCON 2010.
[3] H. Busch, “Formal Safety Verification of Automotive Microcontroller Parts,” ITG/GMM-Workshop ZuE 2012, Bremen, July 2012.
[4] H. Busch, “Qualification of Formal Properties for Productive Automotive Microcontroller Verification,” Proc. of DVCON 2013.
[5] H. Busch, “An Automated Formal Verification Flow for Safety Registers,” Proc. of DVCON EUROPE, Munich, Nov. 2015.
[6] M. Bartley, J. Grosse, “Verifying Functional, Safety and Security Requirements (for Standard Compliance),” Tutorial, Proc. of DVCON EUROPE, Munich, Nov. 2015.