2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE 10 YEAR ANNIVERSARY

Integration Verification of Safety Components in Automotive Chip Modules

Holger Busch, Infineon Technologies, Automotove Microcontroller Division, Munich, Germany
(*holger.busch@infineon.com*)

***Abstract*—In order to meet the ISO26262 standard for automotive safety, chip designs for automotive applications are furnished with hardware safety mechanisms which provide the diagnostic coverage required for the targeted safety integrity level. For this purpose, pre-designed library components related to a specific safety mechanism for registers are integrated in existing functional RTL designs. Even if the library components have been pre-verified and field-proven in predecessor products, their integration in a new design still needs to be verified from two aspects. Firstly, the regular mission function must not be distorted. Secondly, the implementation of the safety mechanisms must ensure that all corresponding safety requirements are fulfilled. This paper presents an efficient integration verification flow for safety library components which is based on formal verification technology.**

Keywords—ISO26262; safety verification; formal verification; register hardening

I. INTRODUCTION

Hardware safety mechanisms in modules of SoCs for automotive applications are required to yield minimum diagnostic coverage according to the automotive safety integrity level (ASIL) classification[1] of each module. For safety assessment, evidence must be provided that a certain percentage of faults is detected by the safety mechanisms. Moreover, it must be proven that fault reaction time does not exceed specified limits, so that the system can handle random faults and return to a safe state before they cause harm to persons or objects.

For this purpose, fault simulators are used which can exercise a big number of fault scenarios by fault injection in safety-critical elementary parts and provide statistics about fault detection rates. Formal verification can check many if not all relevant fault scenarios of a certain fault class simultaneously in a single property[2], if a safety mechanism by construction is able to detect all these. This is the case for error-correcting codes which ensure mathematically that all single- and double-bit errors in a memory word are detected, and single-bit errors are localized and corrected. Here a single corresponding formal property simultaneously covers an astronomical number of data and fault combinations. If a safety-mechanism does not ensure complete detection of all faults, formal verification can still contribute to statistics if distinct subsets of fault scenarios can be characterized by way of corresponding formalizable constraints.

We assume a safety mechanism to be implemented by way of safety components from a dedicated library for different levels of hierarchy. At the lowest level, a spontaneous non-functional bit-flip first needs to be detected locally in the sub-component which contains safety logic with redundancy and comparison function and the capability to assert an alarm flag. As the number of alarms manageable at system level is limited, locally generated alarms need to be reduced across several levels of hierarchy, until finally a joint alarm is sent to the system level so that the system can react in an adequate way to drive the system into a safe state.

Individual registers to be safeguarded have different local reset and clock-domains. When such ordinary registers are replaced with safety register components, their reset- and clock-inputs must be connected to the same domains as the previously unprotected registers to preserve the original regular register function. When combining alarms, clock- and reset-domain memberships must be observed. At top-level, each joint alarm itself needs to be provided in a specific clock domain. To ensure that no alarms from different domains will be lost, hierarchical collection and propagation of alarms includes appropriate synchronization.

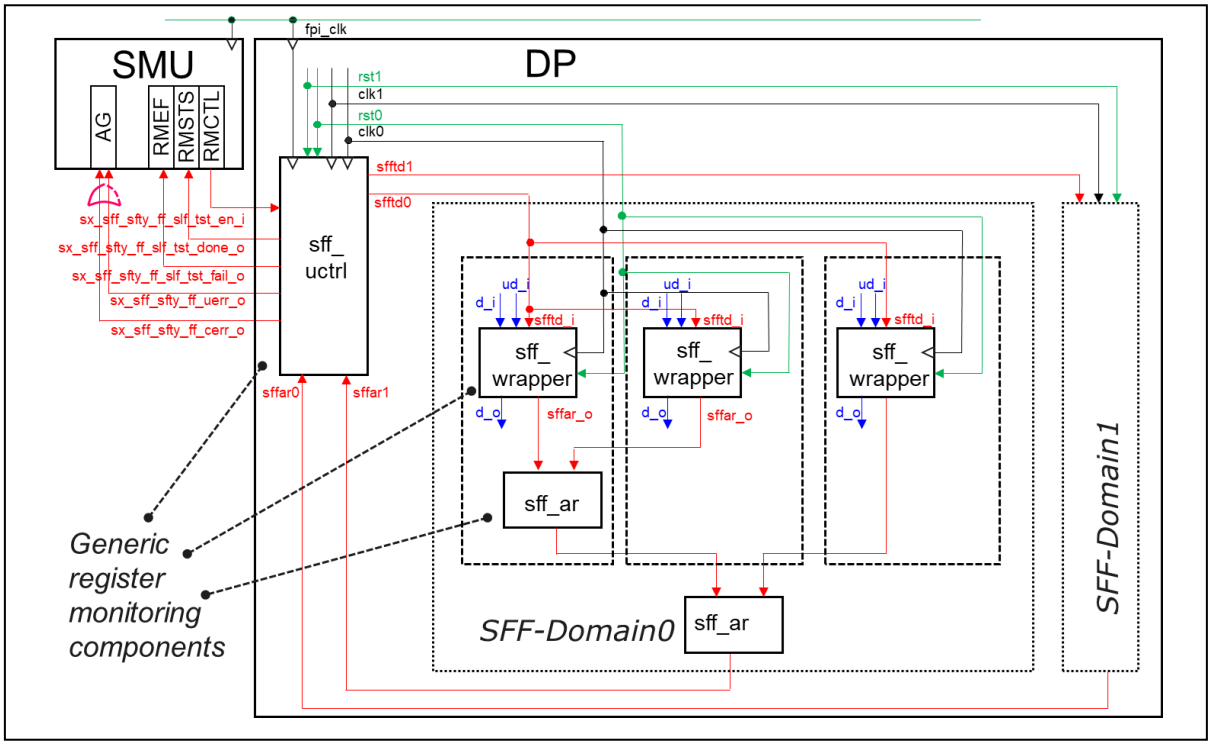
The following sections of this paper are organized as follows. Section II illustrates the basic structure of the safety architecture of modules with register protection. In III, an integration verification plan is sketched. In IV, our verification methodology is described. Experience and results are summarized in V, followed by conclusions in VI.

II. SAFETY ARCHITECTURE

We first give a rough description of our safety-architecture, library components and their function.

Figure 1: Safety-Architecture

The register-monitoring safety mechanism, here also called *SFF-mechanism* (*SFF* for *Safety-Flip-Flop*), is controlled by a *Safety-Management-Unit* (*SMU*), which collects alarms from various safety mechanisms in many modules and triggers individually programmable alarm reactions. For the SFF-mechanism, a dedicated SFF-self-test can be started by software writing to a control register (RMCTL) in the SMU, which then receives a DONE-status signal captured in a different register (RMSTS), and in case of test failure sets a FAIL flag (RMEF).



Three parameterizable library HDL components are used in Figure 1; they have the prefix “SFF”.

A. *SFF-Wrapper*

A register component comprises normal register function and additional encoding- and decoding logic for error detection and optionally correction. Self-test function is available which causes defined bit-flipping and generation of expected alarms. In mission mode, this component behaves like a regular register where each data input bit can be individually write-enabled so that the corresponding register bit is updated in the next cycle and the current value directly driven to the data output.

If the wrapper has been configured with correction, for instance by way of an error-correcting code (ECC), either the output can be directly (asynchronously) corrected, or the corrected value can be written to the internal register so that the correction becomes effective only in the next clock cycle. In the first case of asynchronous correction, the wrapper data-output must not be fed into a synchronization stage without intermediate registration, to avoid potential meta-stabilities.

B. *SFF-Alarm-Reductors*

Several alarms are combined into a common alarm to avoid having to handle them all individually. During self-testing, all input alarms of one SFF-domain are expected to be activated and de-activated simultaneously. The outputs of the alarm reductors are either further combined at higher levels of hierarchy or finally connected to corresponding collective alarm inputs of the umbrella controller. The usage of alarm reduction components is not

mandatory, since each SFF-wrapper could be individually connected to the umbrella-controller. It is recommended though to combine alarms as much as possible before they are received by the umbrella-controller (II.C), in order to reduce wiring complexity through the hierarchy and the number of domain controllers in the umbrella controller.

C. *SFF-Umbrella-Controller*

As all SFF-alarms shall be combined to a top-level alarm forwarded to the SMU in a specific clock domain, the umbrella controller takes care of final alarm synchronization and combination of alarms received from different SFF-domains. *SFF-domain* here denotes the combination of clock and reset input of an SFF-wrapper instance.

Moreover, the controller generates test sequences for the different SFF-domains and in turn collects and evaluates alarms expected in specific self-testing phases in a well-defined way while the self-testing is active.

During self-testing, global alarm and ready notification to the SMU are only sent once when the test-alarms and ready-signals have been received from all SFF-domains. Clock gating or configurable clock division may thus increase the self-test-alarm latency. Nevertheless, whenever a real alarm is generated even in a slow SFF-domain, the global alarm to the SMU is issued without delay independently of the other SFF-domains, because SFF-wrappers and alarm-reductors have no synchronous alarm latency, and synchronization to SMU-clock domain will not cause more than 2 SMU-clock cycles delay. Only in the case that the SMU-clock is configured to low frequencies or even switched off, fault detection times could be longer than acceptable. However, in mission mode, the SMU clock should not be decelerated or sent to sleep mode. If this should happen randomly, there are other system-level mechanisms in place to detect such situations.

III. VERIFICATION PLAN

The verification of the effectiveness of the register monitoring mechanism requires contributions from several verification domains. For each verification domain, a separate verification plan is provided. Globally, it needs to be verified that any fault expected to be caught by the safety mechanism results in an adequate reaction of the system to obviate all failure modes potentially related to the fault.

All library components have already been exhaustively verified including formal fault-injection[3] and have additionally been field-proven in previous product families. The highest remaining risk that the safety architecture might be flawed is caused by manual integration of the library components in the modules. We therefore here focus on the SFF-integration verification plan, which has been kept generic so that it applies to all modules which are equipped with the register-monitoring mechanism. The SFF-integration verification plan consists of two parts referring to configuration parameters, and to connectivity of the library component instances:

A. *Configuration Parameters*

For each library component instance, the configuration parameters must be consistent to the configurations of the connected library component instances. As most modules do not require correction, but just detection of errors, library components can be configured accordingly to save logic gates and reduce wiring complexity. However, their corresponding configurations must be compatible. Moreover, self-testing logic can be omitted by configuration parameters, if other methods yield sufficient test coverage, like LBIST (logic built-in self-test). In all such use cases, the individual library component instances must be configured consistently.

- **Umbrella Controller(s)**:
  For each SFF-domain there must be a corresponding set of alarm inputs and a test-vector output, which can be reduced if no correction is required in a module. A specific configuration parameter allows for each SFF-domain to be specified how the domain clock is related to the SMU, i.e., whether the domain clock is identical, divided, multiple or asynchronous. This clock relation decides upon the automatic generation of the synchronization logic in the controller during design elaboration.
  If there is a hierarchy of several umbrella controllers, certain rules must be observed. For instance, there must be one top-most master for the communication with the SMU and each slave must be connected to one superior controller. Each umbrella-controller can have any number of subordinate umbrella controllers, or none. Most modules have just one umbrella controller, but there are exceptions.

- **Wrappers**:
  If self-testing support is not configured in the superior controller, it does not make sense to configure an SFF-wrapper with self-testing logic. Assuming that such configuration would be unintended, at least a warning will be generated by a corresponding check. Conversely, the superior controller would get stuck if a subordinate were configured not to support self-testing, since it would not generate a test alarm.

These examples illustrate that the configuration parameters of the different library component instances need to be checked for consistency.

B. Connectivity

As can be seen in Figure 1, only the wrapper components have data inputs and outputs connected to regular function, which also determines their reset and clock domains. The other SFF library components do not interact with mission function. The alarm reductors are combinatorial and have only alarm connections on in- and output side. The umbrella controller(s) are connected to the SMU or a superior umbrella controller, and possibly to subordinate umbrella controllers. Each umbrella controller gets SMU-clock and application reset, which is the weakest. In the following, we give examples of connectivity rules specified in the SFF integration verification plan.

- **Umbrella Controllers**:
  The signals from and to the SFF-umbrella controllers control the self-test function in all SFF-wrappers of all connected SFF-domains and collect the reduced alarms from these. Example checks look as follows:
  - Check that the clock input shall be connected to the ungated SMU-clock.
  - Check that all SMU-interface signals of the master umbrella controller are connected to the corresponding ports at the module interface wired to the SMU at SoC-level.
- **Alarm Reductors**:
  Alarm reductors are fan-in-connected to SFF-wrappers and previous reductor stages and their fan-outs are connected to the SFF-domain slot of the umbrella controller. Checks like the following are run:
  - Check that each reduced-alarm vector of the input array is connected to preceding reductors and SFF-wrappers of the same SFF-domain.
  - Check that each reduced input sub-alarm-vector of an alarm reductor is connected to another alarm-reductor or directly to a wrapper of the corresponding SFF-domain.

IV. VERIFICATION METHODOLOGY

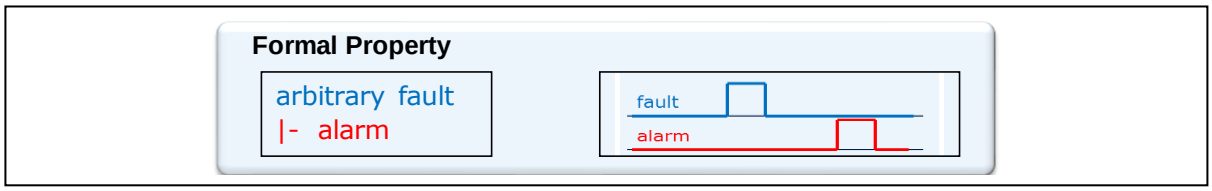
Relying on exhaustive functional SFF-library component verification with formal fault injection, which addresses the fault detection and correction capabilities of each individual component, integration verification does not require functional checks any longer at module level but can be restricted to structural checks.

Structural checks can classically be carried out by providing formal SFF-integration properties and proving them in a formal tool just like ordinary functional properties. We have a flow for generating design-specific formal property code from an elaborated design, inserting such data into pre-defined generic properties.

Such formal properties are logically comparably simple, yet very powerful. As illustrated in Figure 2, the property just assumes an arbitrary detectable fault in an elementary part protected by the safety mechanism and proves that an alarm is raised within limited fault-detection time. Scenarios with resets are excluded by additional assumptions. That property covers a huge number of potentially occurring fault scenarios in a single proof.

However, despite the seeming simplicity, there is substantial computational complexity involved for preparing and proving such a formal property, particularly when big modules are under verification, and when slow clocks are to be handled. Moreover, if the property fails, debugging is normally not easy in a general-purpose debugging environment of a property checker until the root-cause for the failure can be identified. There may be many reasons why an alarm is not flagged as expected in the formal property.

Figure 2: Basic Formal Property for Expected Alarm Occurrence



Most of the time, the root cause for proof failure finally turns out to be simply a missing or wrong connection in the safety architecture. In any case, using a formal property checker and its debugger efficiently requires quite some experience. One of the goals of introducing the safety mechanism in terms of pre-defined library components which encapsulate all error detection and self-testing logic was to relieve design and verification engineers from the need to understand internal logic function of the components. However, a verification engineer without this pre-knowledge easily gets lost when trying to dive into internals of the intricate implementation during root-cause analysis of a counterexample.

All these aspects motivated us to provide an alternative integration verification flow which is not based on formal proofs, but on structural analyses, and which can be used by non-experts including design engineers. These analyses are implemented by automatic scripts which directly check design data extracted from the SFF-architecture for consistency. The most important parts of the flow are summarized in the following.

A. Automatic Design Compilation

We have developed an automatic procedure which locates all information in a release management database required to compile all module instances of a microcontroller product in a formal property checker with all occurring generic parameter sets and dependent libraries. Once the module has been elaborated and compiled, all design data, like instance hierarchy, implementation signals and constants, signal dependencies are accessible and can be further processed automatically for integration verification.

B. Extraction of Design Data

The instances of SFF-library components are detected in the elaborated design hierarchy and their inputs, outputs, and configuration parameters are collected. All clock and reset signals are traced back to equivalent unique signals at highest level in the fan-in of each. This allows us to identify SFF-wrapper instances belonging to the same SFF-domain, but which are spread over many different sub-components of the module architecture.

In the following table, clock numbers uniquely identifying clock signals are shown with parent clock numbers, which have been derived automatically from fan-in-trees and clock-gating logic in the design. A number “-1” is assigned if the clock signal has no parent but is an external module input. By default, external clocks are unrelated.

Table 1: Clocks and Dependencies

no	clk	par_clkL	sel	en
0	inst_smr/inst_ubs_wrapper/clock_reset/clocks__0/clk_gate_1/clk_o	2		sx_dft_clocks_spb_dft_ctrl_i
1	inst_bck_clk_mx/clk_o	3	scan_mode_central_i=0	
2	sx_clocks_clks_max_spb_i	-1		
3	sx_pms_clock_r_clka_100m_i	-1		

If clock-multiplexors and -gates are used, their branching and enabling conditions are also shown in the table. This information is evaluated for deriving the clock relations between two clocks. For instance, if a second clock is derived from a first one by clock gating, they are classified to be related, with the second being slower or equally fast. If an alarm is transferred from the slow into the faster clock domain, it will be captured, whereas in the opposite case, additional measures must be taken to avoid loss of a short alarm pulse in the fast clock domain.

A corresponding table is generated for all reset signals. In the tables for SFF-components, the clock and reset numbers are referenced in place of their signal names, which may have long instance paths.

Table 2 shows data of the SFF-domain controllers within the umbrella-controller. Column clk_diff reflects the relation between the umbrella controller input clock, which normally should be identical to the SMU-clock, and the domain clock. A value 0 means that both are equal, a 1 that the domain clock is divided, a 2 that it is a multiple, and 3 that it is asynchronous to the SMU-clock.

Table 2: SFF-Domain Controllers

no	par_comp	clk_no	rst_no	uctrl_no	dom_no	used	clk_diff	sff_ste_g
0	sff_uctrl	0	1	0	0	1	0	2
1	sff_uctrl	0	2	0	1	1	0	2
2	sff_uctrl	0	0	0	2	1	0	2
3	sff_uctrl	0	3	0	3	1	0	2
4	sff_uctrl	0	0	0	4	1	0	2
5	sff_uctrl	0	3	0	5	1	0	2
6	sff_uctrl	0	1	0	6	1	0	2
7	sff_uctrl	1	4	0	7	1	3	2
8	sff_uctrl	1	4	0	8	1	3	2

Domains with same clock and reset could be combined into one domain, like 2 and 4, or 7 and 8. Then all alarms of SFF-wrappers within this new domain would have to be combined by preceding SFF alarm reductors. In total this would save some domain controller logic in the umbrella controller, but functionally it makes no difference.

Table 3 contains extracted data for SFF-wrappers. Complex modules can have several 100 of these. Column **ctrl_fo** indicates the number of the SFF-domain to which the wrapper is connected, **ctrl_fi** the domain from which it receives the self-test control signal, and **ar_fo** the next alarm-reduction component in the fan-out. Column **SFF_ste** shows whether the wrapper has self-testing logic, followed by columns for data width **dw**, and redundant code width **rcw**.

Table 3: Wrapper Data

no	clk_no	rst_no	ctrl_fo	ctrl_fi	ar_fo	sff_ste	dw	rcw
0	0	2	1	1	1	true	1	2
1	0	1	0	0	0	true	18	5
2	0	0	2	2	2	true	32	6
3	0	1	0	0	0	true	7	4
4	0	0	2	2	2	true	1	2
5	0	1	0	0	0	true	8	4
6	0	1	0	0	0	true	8	4
7	0	1	0	0	0	true	8	4
8	0	3	3	3	3	true	26	5
9	0	3	3	3	3	true	18	5

An additional table is generated from a machine-readable (XML-)specification of special function registers, which contains for each bit-field the information whether and how it shall be safeguarded by the register-monitoring mechanism. In the elaborated design, all these bit-fields are traced to their implementation signals. If these are connected to SFF-wrapper-data outputs, the protection configuration of these is extracted. In Table 4 it is shown that in rows 5-9 there is a discrepancy between what was specified (:= DED for double-error-detection) and implemented (:=NONE). In this case it was not a design bug, but the correct register specification version had not yet been officially released but already been used by the designer.

Table 4 Specification of Register Protection

no	wrp_no	pmeth	sfr	width	sfr_rg	sfr_bf	bf_type	sec_pmeth
0	28	DED	DRVCFGESR0	1	DRVCFGESR0[0]	DRVCFGESR0.DIR	rwh	DED
1	28	DED	DRVCFGESR0	1	DRVCFGESR0[1]	DRVCFGESR0.OD	rwh	DED
2	28	DED	DRVCFGESR0	4	DRVCFGESR0[7:4]	DRVCFGESR0.MODE	rwh	DED
3	28	DED	DRVCFGESR0	3	DRVCFGESR0[10:8]	DRVCFGESR0.PD	rwh	DED
4	28	DED	DRVCFGESR0	3	DRVCFGESR0[14:12]	DRVCFGESR0.PL	rwh	DED
5	-1	NONE	DRVCFGESR1	1	DRVCFGESR1[0]	DRVCFGESR1.DIR	rwh	DED
6	-1	NONE	DRVCFGESR1	1	DRVCFGESR1[1]	DRVCFGESR1.OD	rwh	DED
7	-1	NONE	DRVCFGESR1	4	DRVCFGESR1[7:4]	DRVCFGESR1.MODE	rwh	DED
8	-1	NONE	DRVCFGESR1	3	DRVCFGESR1[10:8]	DRVCFGESR1.PD	rwh	DED
9	-1	NONE	DRVCFGESR1	3	DRVCFGESR1[14:12]	DRVCFGESR1.PL	rwh	DED
10	0	DED	ESR0CFG	1	ESR0CFG[0]	ESR0CFG.ARI	rh	DED

In the table, the column **sfr_bf** contains the specified bit-field type, where *rwh* is used for bit-fields which can be written by internal (*h* - hardware) logic or software, and read by software, and *rh* for a software-readable and hardware-writable bit-field.

C. Checks on Design Data

The extraction data collected in the various tables for each library component instance is sufficient for performing very efficient checks according to the SFF-integration verification plan.

One simple check for instance compares for each wrapper the SFF-domain controller in the fan-in of the self-test-vector, with the SFF-domain controller to which the alarm output of the wrapper is connected. If their numbers are not identical, the domain controller could receive a self-test alarm at an unexpected time, which would result in an activation of the test-failure flag to the SMU.

If in comparison a formal property for checking that without fault injection, the test-failure flag during self-testing is never asserted, were run instead of above check, it would certainly fail. However, debugging the counterexample could well cost a verification engineer without deeper experience with the safety mechanism a lot of time and tedious effort. Even a more knowledgeable colleague would need several manual debugging steps until identifying the wrong connection as the root cause.

In contrast, such structural checks of extracted design data fully automatically and very efficiently detect any inconsistencies in the implementation structure of the safety mechanism and directly expose the bug.

D. Instantiation of Test-Architecture for SFF-Library Verification

For deep library verification, we must ensure that all actual component instances are verified with their specific parameter sets. For this purpose, we use a highly configurable test architecture which can be adjusted according to the extracted configuration parameters of the library component instances in the real module design. The SFF-wrappers in the test-architecture are configured with the parameters from the SFF-wrapper extraction table as shown in Table 3. In these, formal fault-injection makes sure that for all relevant data-widths, the corresponding encoding and decoding logic allows all faults to be detected as specified. The test-architecture is equipped with the reduction components and umbrella controllers with same configurations like in the real module architecture, using the corresponding extraction tables.

V. EXPERIENCE

Compared to previous functional integration verification flows, our flow has substantial advantages. In the past, we had a formal verification team for centrally taking care of the verification of the safety architectures in the context of the full modules. Since then, many more modules have been furnished with this safety mechanism, so that it became necessary to decentralize integration verification.

This has been achieved by splitting the integration verification flow into

1.) a functional part with deep formal verification of library components with exhaustive formal fault injection in a reduced test-architecture, which contains all library components with their actual configuration parameters and connectivity, but no functional mission logic of the original modules, and

2.) fast structural checks on the real module evaluating automatically extracted data of the safety architecture.

Part 1) much less depends on individual module complexity due to the reduced test-architecture without mission function. Part 2), which addresses the practically much more error-prone manual integration of the fully verified library components, is now handled by very efficient automatic, and easy-to-use routines.

In this way we have been able to handle all modules of our microcontroller product derivatives within few days. In fact, one of the more difficult parts was to refine the automatic compilation procedure until all modules with all required parameters and sub-libraries were handled, before running the extraction procedure.

The overall extraction of all data needed for verification and statistics generation takes just few minutes. The most expensive pieces are fan-in and fan-out construction at bit-level, due to the sheer number of bits to be traced in big modules. For this purpose, we have created very efficient sub-procedures for fan-in and fan-out tree construction which exploit pre-knowledge about our proprietary library components. Automatic location of bit-level implementations of special-function-register bit-fields is another more expensive task to generate specification tables as shown in Table 4.

VI. CONCLUSIONS

Compared to conventional functional verification flows, our dedicated integration verification approach has substantial benefits:

1.) Human resources minimization:
The work-split between deep formal library verification to be centrally performed by a much smaller expert team on the one hand, and on the other hand non-experts de-centrally executing the automatic structural flow with negligible effort reduces the overall resource consumption. Using the same automatically extracted data for the automated integration checks and at the same time for populating the generic test architecture according to the configurations within the real modules, we also enable automation of major parts of deep formal library verification.

2.) Run-Time Optimization:
Purely structural checks based on automatically extracted design data which exploit specific structure and characteristics of pre-defined library components are orders of magnitude faster than any general-purpose functional verification approach, be it formal or simulation based.

3.) Exhaustiveness:
All library-component instances are covered including their connectivity.

4.) Easy Usage:
Since no verification knowledge is required to run the automated structural checks, any verification or design engineer can verify correct integration of the library components. Hence, design engineers can run the checks immediately when they have just installed the safety architecture in their modules and can correct flaws very early before any verification engineer gets involved.

5.) Flow Qualification
With the option to generate formal property macros from the extraction data and prove generic properties instantiating these on the modules to confirm the automatically achieved integration check results, we can additionally safeguard our automatic integration verification methodology by a second redundant flow. If bugs have already been corrected due to structural findings, the formal properties can be expected to hold and, in this way, cause no human debugging effort.

REFERENCES

[1] ISO 26262 Std. Road vehicles, Parts. 1-10, 2011.

[2] H. Busch, “Formal Safety Verification of Automotive Microcontroller Parts,” , ZuE’12, 6.GMM/GI/ITG-Workshop, 2012.

[3] H. Busch, “Automated Safety Verification for Automotive Microcontrollers,” in Proc. of DVCON US 2016.