2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

An Automated Formal Verification Flow for Safety Registers

Holger Busch, Infineon Technologies AG, Neubiberg, Germany (*holger.busch@infineon.com*)

***Abstract*—Automotive microcontrollers support an increasing number of safety applications and include various safety mechanisms in order to meet the ISO 26262 standard. Derivatives of product families address different market segments and customers with individual safety requirements. Thus flexibility is needed to adjust the overhead introduced by extra safety circuitry to the actual configurations and safety requirements of the individual product. This paper presents a comprehensive formal verification methodology for safety-critical registers. Distinct sets of a relatively small number of black-box and white-box register properties, which call macros generated from XML register specifications with safety tags, allow the depth of automatic register verification to be progressively increased by inserting more and more design-specific data. This systematic and highly automated formal verification approach, which includes formal fault injection, not only saves functional and safety verification efforts, but also yields evidence for ISO 26262 compliance.**

Keywords—formal verification; special function register; safety register; safety verification; ISO26262

I. INTRODUCTION

The automotive industry is faced with a growing demand for new automatic systems with sensors and microelectronic components which take over more and more control previously under responsibility of the car driver. With more automatic driver assistance systems, the human driver is not only relieved from the need to be fully attentive throughout the whole car trip, but has less chance to react adequately and in time in case of technical mal-functions with the potential to endanger human lives. On the other hand, chip technologies shrunk to structures of 40 nm and below increase the likelihood of random failures e.g. caused by alpha radiation. Therefore, the robustness and reliability of electronic control units in cars have become absolutely vital and need to be ensured by the complete development and manufacturing processes and during the full lifecycle of the cars.

For this reason, the ISO 26262[1] standard was introduced at the beginning of the year 2012. It obliges all car manufacturers and their suppliers including the chip providers to develop new electric/electronic systems for mass-produced passenger cars up to 3.5 tons in compliance to safety requirements in 130 work products of the standard. They cover safety management, the whole safety life-cycle process from concept, development (system, hardware, software) phases according to a V-model, production and operation until decommissioning, and supporting processes like defining responsibilities, configuration and change management, or documentation, and safety analyses according to 4 automotive safety-integrity levels ASIL A (lowest) - D (highest).

While the principles and processes mandated by the standard are not completely new, higher costs and efforts are generated in any case by the augmented development flow and increased system complexity due to new features and safety measures which consume a lot of compute power and additional hardware for safety.

Systematic specification, design and verification processes are crucial to master the additional complexity, and to limit the development costs by replacing tedious and error-prone activities with automatic procedures and maximum re-use of previous results. An indispensable prerequisite of any automation is the accessibility of appropriate input data in well-defined formats. Traditional textual specifications to be manually translated into machine-readable form are not well-suited. Since more and more specification data is being provided in standard formats like XML (Extensible Markup Language), verification automation is progressing significantly. Specifications of special function registers (SFRs) accessed from system busses with well-defined protocols were early identified as good candidates for formalization and automatic processing, which have been extended considerably since then.

Here is an overview of the next sections: In Section II, background information on the field of application and on related work is given. Sections III and IV present our approaches for general special function register verification and for safety register verification, followed by applications, results (V), and final conclusions (VI).

II. BACKGROUND

The work presented in this paper augments and combines two of our formal verification approaches in order to achieve an ISO 26262 compliant formal verification flow for safety-critical special function registers.

We developed a first automated formal verification flow for software-accessible special function registers (SFR) of digital system components a few years ago[5], when safety was not yet such a major issue for microcontroller design, but nevertheless high functional quality was demanded. While property and proof management had already been highly automated, the maintenance of the source data base with register specifications required a few manual activities in order to keep track with different versions of module specifications and designs for varieties of product derivatives of microcontroller families. In the meantime, advanced version and change management methodologies, and various formal verification approaches have been added, some of which are outlined in this paper.

Driven by the need to fulfill a growing number of safety requirements and attain highest ASIL-levels demanded by the automotive industry and assessed by certification authorities, various safety mechanisms were added, which however generated new complexities and verification challenges. Therefore we devised another automated formal verification flow[6] to address soundness, effectiveness, robustness, and integration of dedicated mechanisms for hardening vital registers. Initially the corresponding verification procedures were predominantly fed by design and manually added data, as there was a fairly lose and informal coupling between safety concept and the safety measures implemented by the designers. An improved specification flow provides more machine-readable safety-relevant data which enables new automated safety checks.

We devised the first formal safety verification approach during the development of Infineon's microcontroller family called AURIX™. The AURIX™ architecture was for the first time developed according to an audited ISO 26262 compliant process and meets ASIL-D on an application level. The AURIX product family integrates up to 3 Tricore CPUs + with extra 2 cores in TriCore™ running inlock-step with clock cycle delay. The cores operate at up to 300 MHz clock frequency, and offer up to 8MB memory. Additional safety features comprise safe internal communication busses and memory protection mechanisms. Multiple safety applications such as steering, braking, airbag and advanced driver assistance systems are supported by one unified platform.

A. Formal verification tool environment

Our formal safety verification methodology is based upon, but not restricted to the property-checking environment from Onespin Solutions. We routinely use it for full-fledged IP-level functional verification of a wide range of automotive microcontroller modules. Onespin's tool suite comprises support for several property languages like PSL, SVA and ITL (proprietary: Interval Language), a set of different provers for specific classes of proof problems, a GUI with debugging functionality, counterexample and witness generation, RTL code linting, and automatic consistency checks with dead-code analysis and others. The completeness of property sets with respect to the full input-/output behavior can be proven by Onespin's formal completeness checker[4]. Another function called Quantify computes structural code coverage which not only includes statement and branch reachability, but also observation coverage. A link to Synopsys' Certitude tool [6] allows the same metrics classically used for simulation-test bench qualification to be applied to formal property sets. For safety qualification such coverage and completeness features are crucial. Onespin's TCL shell provides a set of user-accessible functions used for simple automatic standard checks ("APP"s), and which allow advanced users to create own dedicated proof automation procedures. The proof functions described in this paper are based on such Onespin utilities together with numerous proprietary ones.

B. Related work

Several formal-tool vendors like Cadence (IFV, Jasper Gold), and Onespin, offer standard register verification APPs which support the IP-XACT[3] standard initiated by the SPIRIT consortium[2]. Our developments started earlier and have since then been continuously augmented, always driven by concrete needs of Infineon's different automotive microcontroller families. Our tooling is oriented towards machine-readable specification formats provided by our concept engineering, and we claim to have a much richer dedicated register verification infrastructure than external register verification APPs addressing a broader market can offer.

III. FORMAL REGISTER VERIFICATION FLOW

A. *Overview of automated register verification flow*

In Fig.1, our XML-based special-function-register verification flow is sketched.

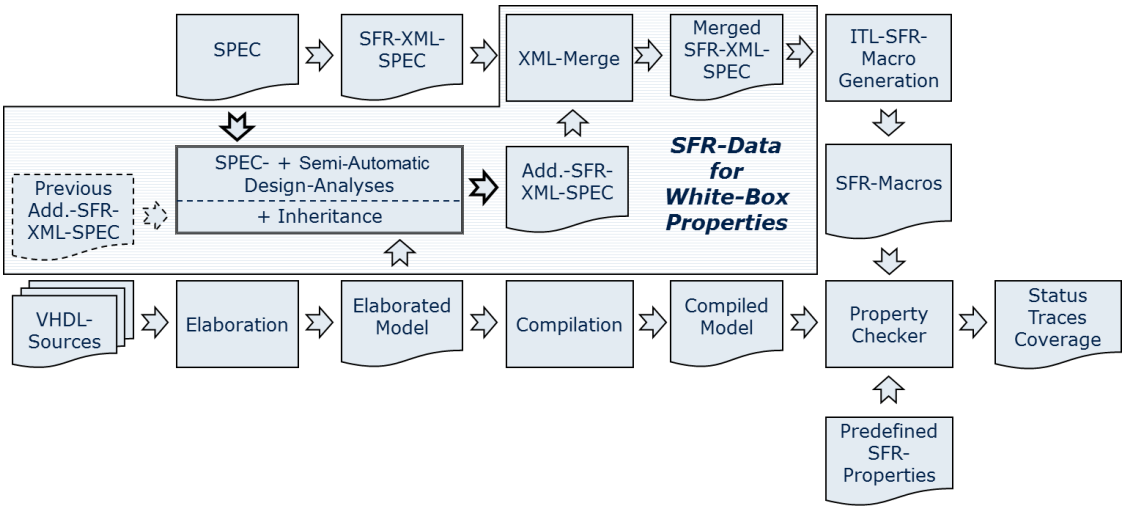

Figure 1: Formal register verification flow

The concept engineers automatically generate XML files with SFR data from their specifications. This information is largely sufficient for black-box checks. For white box-checks, another XML file (Add.-SFR-XML-SPEC) that is partly extracted automatically from the elaborated model of the RTL design (like hierarchical signal names), and partly enriched manually with more register information (e.g. internal write conditions, cf. Paragraph C) not explicitly available in the automatically generated standard files (SFR-XML-SPEC) is added. In order to minimize the manual editing work, we have added an inheritance mechanism which allows us to re-use data pertaining to registers and bit-fields already comprised in prior module versions. Of course this data and the corresponding piece of specification still need to be reviewed, but this costs much less effort than editing again from scratch. A specific XML-merge function merges all relevant XML-specifications into one joint XML file (Merged SFR-XML-SPEC) in which all data of each register bit-field is combined in one place. Different bit-fields with same name in one register are automatically uniquely renamed (typically reserved or unused bit-fields, e.g. named "0"). The enriched XML-specification is fed into a generator for all required macros in the property language. Pre-defined generic register properties then include these macros. If before XML-merging no design-specific XML file is available, the macros are nevertheless generated with default definitions based on heuristics which in many cases produce meaningful results.

A strict release management of all specifications, designs, verification plans, et cetera, ensures that the versions being used for property generation and formal verification match. Thus there is no risk that a concept engineer delivers an updated specification with neither design nor verification engineer noticing it.

B. *Black-box properties*

This group of properties can be applied without knowing any internal signal names. If the register implementation in terms of hierarchical design signals is not yet known, register checks can only be performed via the bus-interface, and register contents only be accessed by a read operation. At least specific macros abstracting bus signals need to be instantiated, otherwise even black-box properties are not provable. This is normally no problem, as the bus interface signals can easily (and most often automatically) be identified due to generally prescribed bus-signal naming conventions.

XML-tags provided by concept engineers specify SFR data at 2 levels:

- Register level: register name, address, length, write and read access restrictions, reset value, reset class, reset mask
- Bit-field level: bit-field name, index range, bit-field type

Correspondingly, black-box properties allow e.g. bus protocol violations to be checked, by assuming a write or read access with active protection, and proving that a bus error is returned. By reading an addressed register before and after an **illegal write** access and additionally assuming that no intermediate legal write to this address occurs, it is additionally verified that the register contents are not modified by the illegal write. **Legal writes** are verified by checking that a subsequent read returns the previously written data value. Specific operations like **byte**- or **half-word writes**, or **read-modify-writes** are verified in a similar way. **Reset** properties need the information about which reset signal at the module interface is related to which reset class. As the names of reset signals also follow strict naming conventions, automatic generation functions do the instantiation of the generic reset signal macros used in the properties. In order to check the correct reset value, the corresponding reset property assumes that a read operation is performed after the reset without intermediate write. As all these operation properties use the address macros instantiated with XML-data, they also verify that the register is accessed by the specified address. An **idle** property verifies that without intermediate write between two reads, the contents of a register are not modified.

For modules with a large number of standard registers with read-only or software-write-read-bit-fields only, a good portion of the register function can be verified by such black-box properties, which in the best case require no user input, and can then easily be run by verification engineers knowing neither the module design, nor the formal property checking environment.

Registers with other characteristics may still be partially coverable by black-box properties which only check the standard bit-fields. For this purpose, a masked-equality macro is used in the black-box properties which allows the check for equality between expected previous write ($wval$) and actually seen read value ($rval$) to be restricted to individual bits or bit-slices by a mask parameter which causes only its 1-positions to be checked.

$$maskEq(rval, wval, mask) \coloneqq rval \; and \; mask = wval \; and \; mask \tag{1}$$

The mask parameter is for example instantiated to a bit-field mask which has ones only in the index range of an individual read-writable bit field of the currently checked register.

C. White-box properties

All bit-fields that are not read-accessible, and such ones which can be updated by hardware, or which have additional internal write protections, cannot be covered by black-box properties. White box-properties need macros generated from additional XML-tags (Add.-SFR-XML-SPEC in Fig. 1) often not provided by concept engineers, such as:

- Register level: hierarchical register implementation signal name, additional reset classes and corresponding reset values
- Bit-field level: write latency, hw-sw-write priority, hw-write conditions(s), hw-write-value(s), default value in case of no write, local sw write protection condition, local reset class

Not all of these white box specification items are needed for all white box checks, and some of these can be inferred automatically from others. For instance, the write latency of any software-updatable bit-field is given by the global write latency configured in the bus-interface. A hw-only writable bit-field naturally has no software write protection. Other data is derived from automatic fan-in analyses in the elaborated design model, such as a local bit-field reset class if it deviates from the global register reset class. All information extracted from the RTL needs to be inspected and matched against informal descriptions in the specification, common understanding, or discussed with the responsible concept engineers.

The hierarchical signal name of the register implementation is the most important item needed for all white-box checks, as it allows the current state of the register to be directly accessed, instead of a more expensive read operation. Thus the proof times of white-box properties are typically smaller because of the shorter examination window. If well-defined naming conventions are followed, corresponding functions for filtering the implementation name of the register automatically from all hierarchical signals can be applied.

Internal software write protections can often be derived from bit-field specifications, as for example the setting of a lock-bit visible in a different register bit-field or a set-only or clear-only restriction.

Hardware-software write priorities are often either not explicitly specified, or just mentioned in the textual bit-field description. Hardware update conditions and values are typically derived from textual bit-field specifications and corresponding pieces of logic in the designs. If the logic expressions contain too many implementation details and more than just few basic logical operators, it is recommendable to introduce macros with telling names which hide these details and provide a higher-level of understanding independent of property language syntax.

Reset properties have two aspects: the first check is to verify that the activation of a reset corresponding to the specified reset class of the current signal actually initializes the register to the specified value. The second negative aspect is more difficult to verify: another reset different from the given reset class must not clear the current signal contents. This independence check is performed by a specific bit-level white-box property:

$$arst \wedge next(arst) \wedge next(changed(reg(i)))| - true \quad (2)$$

There are two cases: 1. *reg(i)* is in the *arst* application domain or 2. it is not. In Case 2, the bit might be specified to be in the power-on reset domain and must not be cleared by an application reset (arst). Assuming the reset condition in both consecutive time steps ensures that any state signal belonging to the given (here asynchronous) reset class is bound to its reset value, and thus cannot change from one to the next time step. In this way, the assumption of the property in total is empty, which will be indicated by the Onespin-provers as *vacuous* result. Hence a *vacuous* result means that no witness exists where the reset occurs in two time steps while the register changes from or to a value different from its reset value. If the register bit is not affected by the reset, the assumption is non-empty, because the state can be changed by at least one write condition (unless the register is trivially constant which is excluded otherwise to be a non-state signal), and a hold is yielded as proof result. A black-box property could not yield a reliable result, as a following read operation would be required, which however would be affected itself by the application reset. Even if the register were entirely located in the power-on reset domain, a hold-result of the corresponding property at register level would only prove that at least one bit of the register is not in the application reset domain. Therefore it is necessary to do this check white-box and at bit level. Without the feature of distinguishing *vacuous* proof results, the *hold* case of (2) would have to be replaced by an expected *fail* result of a positive reset check:

$$arst \wedge next(arst)| - \ next(stable(reg(i))) \quad (3)$$

If this property fails, the register bit obviously cannot belong to the tested reset domain. We prefer to include only properties with expected *hold* result in our property regression suite.

D. Concept of aggregate register properties

In previous work we introduced a concept of register lists and corresponding aggregate properties which allow formal checks of all registers and all their bit-fields to be performed in one single proof. The property macros are written in such a way that they take arbitrary lists of registers as parameter, i.e. all SFRs, subsets, or just a single SFR. By way of typical list operators implemented in the property language, we can also exclude registers from the aggregate checks. The huge benefit of this aggregation is that the regression run time is reduced by orders of magnitude, due to the fact that the proof time for an aggregate property covering more than 100 SFRs in one proof is in the same range as for a single proof of just one register. Nevertheless, we have also provided functions for generating and proving instantiated properties for each SFR instead of aggregate properties.

The negative reset check discussed above is the only example where aggregation does not help, because a hold result would only prove that at least one bit of all registers passed as list to the aggregate property is not in the wrong reset domain, which would not be very interesting. As these negative reset properties have very short proof-times, it is not important that they are not aggregated. However, in general proving more than 1000 single instantiated SFR- or even bit-level properties instead of just one aggregate consumes much more computing power and is only done in very rare cases in order to evaluate specific aggregate SFR properties and generate separate counterexamples or witnesses for single SFRs.

IV. VERIFICATION OF HARDWARE SAFETY REGISTER REQUIREMENTS

For safety-critical special function registers it is crucial that contents are not corrupted by unintended software writes or spontaneous random bit-flips. Therefore two groups of safety requirements are typically specified for safety SFRs: write access restrictions, and error detection and correction mechanisms.

The access restrictions are already covered by aggregate register properties discussed in the previous section. Such access restrictions are for instance controlled by watchdog timers started by specific masters for limited time frames during which accesses are allowed. Other access restrictions are controlled by specific registers allowing only configured masters to access the protected masters from unauthorized modification. Some registers can only be modified during start-up like specific product configuration registers. All these requirements are already covered by our regular SFR verification flow discussed above.

The protection of safety registers from non-functional random faults requires specific safety mechanisms based on redundancy. Various approaches are applied for SFR safeguarding: parity bits, double or triple modular redundancy (DMR, TMR), error-correcting codes (ECC) in different version like SECDED (single error correction, double error detection), DED (no correction) or others, depending on the specific needs. Further tags allow the specification of self-testing features. For all these hardening approaches we have specific library subcomponents which allow the hardened registers to be easily identified in the list of all hierarchical signals of the design. Corresponding automatic macro generators construct vectors containing all safety registers and their duplicates, respectively, and difference vectors which locate assumed errors. Few pre-defined properties call these macros which e.g. prove that any selection of n bit errors in redundant bits causes an alarm:

$$be(n), n > 0| - alarm \tag{4}$$

This property is extremely powerful as it allows all possible fault combinations to be covered in one single proof, instead of iteratively assuming a fault in each individual bit. For this purpose we developed a very efficient formal fault injection[6] approach based on Onespin's cut feature which allows signals to be cut into fan-in and fan-out part which are during compilation turned into global output and global input of the module. By providing assumptions on the resulting new global inputs different fault models like negation are usable.

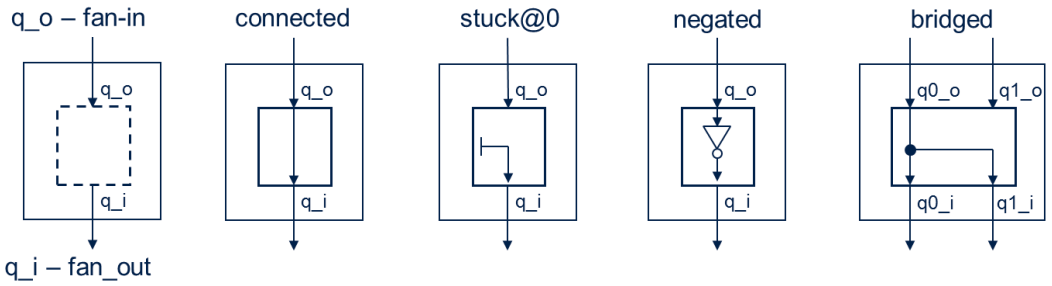

Figure 2: Assuming fault models by assumptions on fan-in and fan-out part of cut signal

The term be(0) then denotes that all cut signals are re-connected, by simply assuming for each cut bit that it's fan-in part is equal to its fan-out part.

A similar property like the one above proves that no alarm is flagged if no bit error occurs:

$$be(0)| - \neg alarm \tag{5}$$

Other properties are related to self-test mechanisms, where a test-enable triggers an alarm, e.g. caused by a defined internal bit-flip:

$$te \wedge be(0)| - alarm \tag{6}$$

Once all regular operation properties are proven, and a register safeguarding mechanism with correction has been installed in the design, all register properties are proven to hold if only correctable errors are assumed.

New XML-tags added to the regular SFR database allow the specification of required safeguarding measures:

- Register level: safeguarding with detection / correction / none, self-testing of safety-mechanism
- Bit-field level: safeguarding with detection / correction / none, self-testing of safety-mechanism

If no bit-field-level safety specification is given, the bit-field inherits the register-level safety-specification. The bit-level specifications allow the safeguarding to be restricted to a subset of the bit-fields of one special-function

register, which may combine safety-critical with safe bit-fields. In so far it is possible to restrict the register safeguarding exactly to bit-fields which are critical in order to minimize the overall redundancy.

Specific verification procedures ensure that the safeguarding approach has actually been implemented as specified. For this purpose additional fan-in analyses are performed. If (combinatorial) correction is included e.g. by ECC-decoding or majority voting, the implementation signal of the register is determined as the combinatorial decoder output, not the registers which are included in the fault-injection complain. Thus for each safety SFR the corresponding internal register data and redundancy bits are identified by these analyses, in order to make sure that the internal register safeguarding matches the XML-specification.

V. APPLICATIONS AND RESULTS

We have successfully applied our automated formal register verification flows to many different system control units with more than 100 special function registers with 32-bits each. In the following, experience and run-times for the most important subtasks are summarized. The proof jobs are sent in parallel to Linux hosts of a load-shared compute farm with hundreds of hosts with e.g. 132 GB main memory and 24 CPUs. As these resources are typically shared with thousands of other interactive and batch jobs, we here give figures in terms of net CPU times and memory usage reported by Onespin's property checker. By selecting the provers in regressions according to previous runs, we are able to minimize the number of parallel threads for each individual property proof to 2, and limit memory to 50 GB.

A. Property-macro generation

The runtimes for transforming 17000 lines of XML code for a module into a comparable number of property code lines by a combination of PERL and TCL scripts just takes several seconds, thus they do no matter at all. Reading the generated macros and predefined generic register properties into Onespin just takes few minutes.

B. Aggregate register property verification

Our SFR regression suite comprises about 40 aggregate black-box and white-box properties in total, most of them are proven in few minutes, the longest needs 39 minutes. These times can differ for the same property with each run depending on the dynamic resource sharing with parallel jobs on the hosts from and to which a proof job is submitted. The register properties typically already yield 50% observation coverage, which is a good basis.

C. Safety properties without fault injection

About ten aggregate safety register properties are mostly proven in few seconds, the longest in less than one hour. The property set includes checks of a dedicated self-test mechanism with alarm activation by defined flipping of all safety register bits.

D. Safety properties with fault injection

We have defined three groups with four properties each, assuming no errors, one-bit errors in arbitrary positions, or any number and any arbitrary combination of detectable bit-errors. The longest proof takes two hours, whereas most of the proofs just need few minutes. Given the astronomical number of fault combinations for more than 3000 safety flip-flop-bits, the formal safety verification proofs are extremely efficient.

These proven properties yield the required evidence for ISO 26262 certification that the installed dedicated safety mechanisms for registers work correctly and provide 100% diagnostic coverage of all register bits. In fact, during the development of these register safeguarding mechanisms, formal verification was applied right from the beginning, therefore it was not a surprise that they worked correctly in the context of bigger modules. Few bugs were discovered during formal safety verification. They were due to wrong integration or incomplete inclusion of local alarms in the global alarm reduction logic. For this purpose, formal fault injection was found to be important as it is able to detect any fault not leading to an alarm.

VI. CONCLUSIONS

Compared to few years ago, the complexity of automotive microcontroller development has substantially increased due to many additional features, and more and more memories and CPUs, by safety measures having been installed on top of regular functionality, and by the requirements of the ISO 26262 standard regarding the development process. Evidence must be provided that all applicable work packages of the standard have been

fulfilled, which is supported by full functional coverage. At the same time, cost and schedule restrictions have become even tighter. This conflict can only be resolved by a systematic verification methodology, with maximum re-use and automation. Our approach for fully verifying safety-critical and safe special-function registers offers these benefits at maximum flexibility for covering any derivatives of microcontroller product families:

1. Highest quality by exhaustive formal verification of all SFR functionality.
2. It is and can be further tailored to the exact needs and specifics of individual modules of our automotive microcontroller systems, which could only be partially achieved by general pre-defined fixed register flow based on IP-XACT.
3. It allows initially shallow register investigation to be gradually deepened by adding register data from the design. By inheritance of register data from previous products, it minimizes efforts to real changes, such as new SFRs or new bit-fields in re-used SFRs.
4. It takes into account all register safety requirements and verifies their sound implementation.
5. By covering all regular and safety functionality of all registers, it gives evidence for ISO26262 certification that all measures are effective according to the claimed ASIL-level.

The set of different features of system control registers is more diverse than of registers in most other modules. The presented register flow handles all these features, and can be easily augmented if needed.

Our formal register verification flow has meanwhile been used productively in the development of several product families. Driven by new safety requirements and the ISO 26262 standard, we have continuously augmented this register verification framework with more general-purpose and specific safety-checks related to new implemented register safeguarding mechanisms, and connected it to our release management system. The high degree of automation allows the huge amount of register data to be maintained and to be kept consistent to the specifications, which is particularly important for product families with many derivatives with varying special function register sets and bit-fields. Since new modules always have a significant portion of re-used functionality, which is also demanded by customers who need software compatibility, our inheritance approach for passing unchanged register data through to new versions where possible saves a lot of human effort. A specific XML patching procedure allows us to perform pre-verification of design and specification changes before their official release.

We believe that formal verification is the ideal approach for register verification, because formal register property sets cover all register functionality including corner case scenarios with any conflicting simultaneous accesses which would be difficult to trigger by simulation. The aggregate register properties contribute a good deal to the overall coverage of a module. The separation of black-box and white-box checks allows fast formal register checks at low effort for other modules which are normally not formally verified.

Future improvements will address a better combination of specification- and design-related data in a common single-source database, which will enable not only the generation of verification, but also of design code. As any hardware safeguarding costs additional area and power, we also think of methodologies to support the minimization of redundancy by formal property checking. Finally, a closer integration of formal verification results with safety-verification at gate-level has the potential to reduce the quantity of fault-simulations. We feel that is should be possible to give a rationale in the safety certification process that a safety-mechanism with provably 100% diagnostic coverage requires just a small amount of fault-simulations at gate level.

REFERENCES

[1] ISO 26262 Std. Road vehicles, Parts. 1-10, 15 Nov. 2011.

[2] IP-XACT™ User Guide v1.2, SPIRIT Consortium July 2006.

[3] IEEE Std 1685-2009 for IP-Xact, “Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows”.16 Feb 2010.

[4] M. Siegel, ”Verification Coverage and Productivity Through Formal Operation- and Transaction-Level Verification Using SVA”, Tutorial at DVCON 2010.

[5] H. Busch, “Generation of Complete Aggregate Formal Properties”, in Proceedings of DVCON 2008.

[6] H. Busch, “Formal Safety Verification of Automotive Microcontroller Parts,” ITG/GMM-Workshop ZuE 2012, Bremen, July 2012.

[7] H. Busch, “Qualification of Formal Properties for Productive Automotive Microcontroller Verification,” Proc. of DVCON 2013.