Automatic Insertion of a Safety Mechanism for Registers in RTL-Modules

Holger Busch, Infineon, Munich, holger.busch@infineon.com
Jonathan Ross, Infineon, Munich (Student Employee from TUM), jonathan.ross@gmail.com

***Abstract*- According to the automotive ISO-26262 standard, random bit-errors in registers controlling safety-critical function in modules of microcontroller SoCs for vital applications need to be detected. Dangerous failure modes are to be caught by taking adequate and timely reactions and driving the system into a safe state before potential harm is caused to human lives. Using a library of formally pre-verified parametric safety components, a specific register monitoring mechanism has so far been manually integrated in many modules of automotive microcontroller product families of Infineon. The integration of these library components must follow certain well-defined rules, which is ensured by dedicated integration checks of the modules. We now have taken the next step: Having demonstrated that the integration verification of this safety mechanism can be automated, we felt that the previously manual integration process itself should also be automatable. This paper summarizes how analysis and verification routines are enhanced to fully automate the installation of the safety mechanism in such a way that not only substantial manual design effort is saved, but also correctness by construction is achieved.**

I. INTRODUCTION

Since the first generation of the AURIX microcontroller family from Infineon, we have integrated a register-monitoring mechanism in many safety-critical modules which have to fulfill the requirements of automotive-safety-integrity levels (ASIL) according to ISO 26262[1]. Examples are system control modules like reset- and clock-control units and memory controllers. For this purpose, we have developed and continuously enhanced a **S**afety-**F**lip-**F**lop (SFF) library of highly configurable safety components (Fig. 1) and specified a corresponding design methodology together with integration rules. From the very beginning, we have supported integration verification by formal-property-checking and more recently by fast structural analysis routines[2]. This has been essential to ensure that the quite sophisticated pre-defined safety components have been correctly installed. However, module designers so far could not enjoy comparable support and automation. They had to study work-instructions and spend manual effort for transforming previously unprotected logic blocks in a sound way so that the safety function was added without corrupting the regular mission function and no alarms could be lost. While the comprehensive post-hoc verification safeguarded the correctness of final design versions before release, bugs could well occur in preliminary versions and cause substantial debugging and correction efforts even in late design phases.

This paper presents an automated methodology which transforms an unprotected register-transfer-level (RTL) design of a module or design part (DP) into a safeguarded version preserving all previous mission function. Additional function is integrated for error detection and optional correction, which also includes a hardware self-test mechanism that can be started at any time by customer software without interruption of the regular operation. The resulting safety architecture consists of inserted instances of the formally- and field proven safety-library components and generated wiring between these. These components encapsulate all register function, alarm- and self-test-logic, so that designers don't have to re-invent similar solutions. Moreover, the automatic insertion procedure avoids potentially suboptimal implementations adding more redundant area than necessary, even if being functionally correct. A preliminary study was done in[3]. We chose the script language TCL[4] for implementation, because the used formal-property-checker environment[5] provides a TCL-shell with basic utilities for accessing data of compiled designs like constant-values, signal-fan-ins and much more. Moreover, proprietary routines we developed for our integration-verification flow[2] are re-usable for design transformation. They could be re-implemented in other languages and tool environments, provided these support comparable design data extraction function. Some routines have been written in Python[8].

This paper is organized as follows. In Section II, three basic library components of our hardware safety mechanism are recapitulated. Section III roughly describes the manual methodology for integration of this safety mechanism in unprotected designs. Section IV sketches automatic sub-routines contributing to structural integration verification. We indicate how some of these are used for design analyses required for setting up a sound safety architecture based on the library components. In Section V, sub-routines for transforming original register logic are explained. In Section VI, everything is put together to our coherent automated register hardening flow. Section VII discusses results and gives an outlook on extensions for the next future.

II. Library for Register Protection

Three parametric library components for installing our register monitoring hardware safety mechanism are offered: a register wrapper, an alarm reductor, and an umbrella controller. Internal SFF-library component details not relevant for the automated flow are neglected.

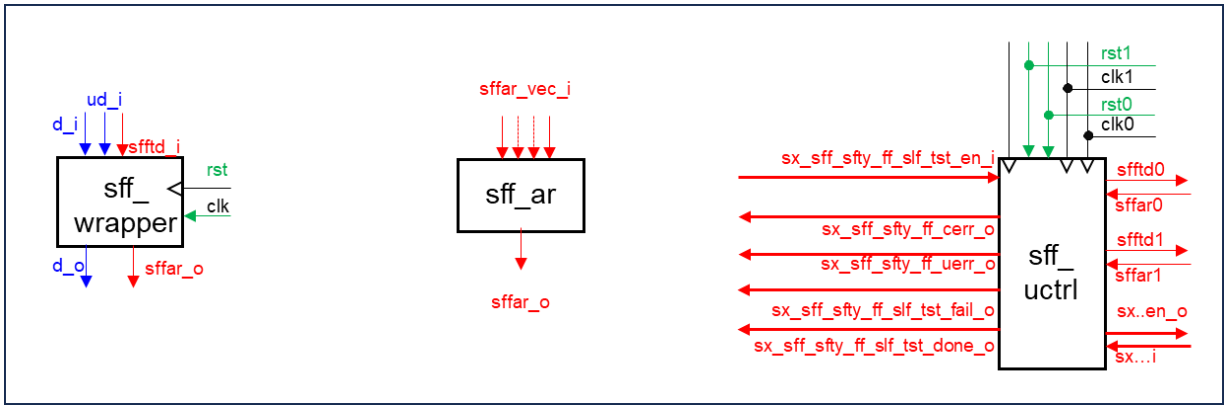

Figure 1: SFF-Library Components

A. *SFF-Wrapper*

The wrapper contains regular register function and configurable redundancy which allows random faults to be detected. It can be configured with a self-test feature for flipping redundant, and optionally also data bits in a defined way to make sure that the alarm logic and propagation works and to detect stuck-at faults. For this purpose, the wrapper has an extra test-input vector (sfftd_i) which encodes self-test phases. In each phase, bits are flipped according to a specific protocol so that at the end of a test sequence the register contains the correct functional value. In each phase, alarm signals behave in a pre-defined way, so that any deviation indicates malfunction of the safety mechanism.

The wrapper has a data input vector (d_i) and an update vector (ud_i), which enables bit-precise writing of new values. Each bit of the register can be configured to be combinational or sequential, unprotected or protected, testable or non-testable. Another generic parameter allows the protection method to be chosen, such as double or triple modular redundancy, parity, or ECC protection. Implementing register function, the wrapper has a reset and a clock input. Alarms are generated by combinational logic in case of any discrepancy between productive and redundant data.

B. *SFF-Alarm-Reductor*

To reduce the number of domain alarms to be processed by the central controller, alarm signals from different wrappers with same clock and reset domain (:= SFF-domain) can be combined before being propagated.

C. *SFF-Umbrella-Controller*

All control function regarding alarm collection and test-phase generation is encapsulated in a highly configurable umbrella controller which includes a separate domain controller for each SFF-domain.

Apart from synchronizing and combining alarms received from the different SFF-domains, the umbrella controller generates the test-phases to the domains, which potentially requires synchronization in the other direction. The generation of synchronization logic is configured by way of a generic parameter which depends on the clock relations between the ultimate alarm to the Safety-Management Unit (SMU) and the local domains.

III. Manual Integration of Safety Components

For automating a manual methodology, its individual steps need to be carefully analyzed. Manual integration of the register monitoring mechanism is recommended to be carried out by first replacing original registers with SFF-wrappers, then inserting SFF-alarm-reductors and finally installing an SFF-umbrella controller, and wiring everything together. In the following, the basic steps for integrating the library components are summarized, which comprise insertion and configuration of these, and wiring them through the hierarchical module architecture.

A. *Insertion of SFF-wrapper*

1. Extraction of the next-state logic into a combinational process for assigning the write value and generating the write strobes at bit-level for overwriting the previous register value.

2. Providing conversion functions of data-inputs and -outputs of the SFF-wrapper between the original register datatype and the bit-vector type expected by the SFF-wrapper.
3. Inserting an SFF-wrapper, connecting its data and update input vectors to the converted signals.
4. Connecting the re-converted SFF-wrapper data output to the fan-out of the previous register. The approach is here just to give the converted data output the same name of the previous register so that the RTL code of the register fan-out logic needs no update.
5. Configuring protection method and self-test-support of the wrapper according to the specification table.
6. Configuring bit-mask constants which decide upon the protection of each individual register bit according to the specification table. This approach particularly allows different bit-fields of software-accessible **s**pecial-**f**unction-**r**egisters (SFR) to be individually chosen to be protected or not, depending on their safety-criticality. Unused bits of an SFR can additionally be masked away so that no superfluous flip-flops are allocated for them and there is no necessity that the design engineer maps index ranges of SFF-wrapper ports manually to bit-slices of original vector signals.

B. Insertion of SFF-alarm reductor

1. Identify combinable alarms according to SFF-domains, starting with component instances containing SFF-wrappers, and proceeding with instances at higher levels which get combined alarms from their sub-components, from SFF-wrappers and subordinate alarm-reductors.
2. For each SFF-domain in each component instance with more than one alarm insert one SFF-alarm-reductor, otherwise no reductor is needed.
3. Augment the component instance interface by the required combined-alarm outputs.
4. Connect the SFF-alarm reductor input to compatible domain alarms from SFF-wrappers or from sub-components and wire its combined-alarm output through to the current component interface.

C. Insertion of SFF-umbrella controller

1. In a selectable parent component, insert an SFF-umbrella-controller.
2. Configure it according to the overall number of different SFF-domains, their self-test support and whether they provide error correction or detection only.
3. Connect all combined alarms received for the different SFF-domains to the domain controller inputs of the SFF-umbrella controller. Augment the interfaces of all components in all paths from SFF-umbrella controller to SFF-wrappers and wire the test-control inputs of these to the test-control outputs of the SFF-umbrella controller for the corresponding domains.

The described steps are schematic and in total not very difficult, but their manual execution is error-prone and costs the design engineers quite some effort.

IV. Subroutines for Structural Integration Verification

In [2] we described our automated SFF integration verification flow, which contains elements also useful for design transformation. In this section, such re-usable routines are summarized. Additional background algorithms are invoked to generate area-optimal error-detection logic, taking into account gate-counts and -areas.

A. Fan-in- and -out tree-computation

A proprietary very versatile and efficient recursive routine for determining transitive fan-ins and fan-out of bits or signals can specifically be used for tracing alarm-, reset-, clock-, and test-control signals between SFF-components and global module ports. It takes a bit or a signal as first argument and can work at bit or signal level according to the second parameter. As third parameter a direct-fan-in or -out function can be given, as provided by the tool environment, followed by a number for the maximum recursion depth, which can be unlimited if the value 0 is given. In this case, the recursion is stopped when either a global input, a previously visited, or a member of a termination list has been reached. The complete fan-in tree of a global output can have a depth of several 100 levels. Further parameters specify bit or signal lists for local or global termination of the recursion. Finally, the construction can be selected to be depth-first or breadth-first, yielding different results if global termination is applied. In the context of SFF integration verification., this basic procedure is used for different purposes, like the following one.

B. Determination of global clock and reset domains of registers

Another routine allows us to determine the SFF-domains of the safety-critical registers which have been or shall be protected. Global clock and reset domains are traced from local clock and reset inputs of registers to identify equal

SFF-domains of registers in different module components. By tracing clock signals through clock gates to global clock inputs, we are further able to determine clock relationships. From these we can infer whether synchronization logic and which kind of it needs to be allocated in the SFF-umbrella-controller. For instance, if an SFF-domain clock is asynchronous to the clock in which the global alarm is sent from the SFF-umbrella controller to the safety management unit, synchronization needs to provide a handshake protocol in both directions, i.e. for sending the test phase to the SFF domain and receiving the domain alarm so that no alarm is lost even if the domain alarm pulse is shorter than the clock period of the external alarm to the SMU. The asynchronous clock is assumed if it depends on a different global clock input of the module than the global alarm. Such information is used to check whether a design engineer has correctly chosen the corresponding generic parameter of the SFF-umbrella controller. In the flow for generating the SFF-architecture, it is used to determine this configuration parameter of the umbrella controller automatically.

C. Checking whether registers are safeguarded as specified

A specification of all registers to be safeguarded is given as a table (Fig. 2) where for each register the protection method, e.g., double-bit error detection (DED), and the bits to be protected are specified, and whether it should be self-testable, e.g. by reduced testing (RT: only redundant bits flipped). If this specification table is not yet available, at least a register file will be provided by concept engineering which specifies bit-field layouts and attributes of all special function registers and bit-fields, including their safety properties. A specific sub-routine also used for automatic register verification locates the implementation signals of the special function registers in the RTL design. It works regardless of whether the special function registers have been implemented by SFF-wrappers or ordinary unprotected state signals. One SFF-integration check then compares the specified attributes with the actual implementation.

The specification table contains at least all safety-critical special function registers and specifies protection and self-test method. Ideally, design, concept, and functional-safety engineers have also agreed which internal registers should be protected. By way of structural analyses of internal registers in the fan-in and fan-out cone of safety-critical module ports and registers, we can optionally generate suggestions for the protection of further internal registers. In this case, the SFF attributes of the previous registers are inherited to the additional ones. Regardless of how the specification table has been created, or of whether just the special function registers or additional internal registers are contained, any such specification table will be accepted as input to the automatic register-protection flow.

	A	B	C	D	E
	bit	**pmeth**	**stmeth**	**sfr_bit**	**bit-field**
	inst_scu/inst_ccu/ccu_concon_reg_o(22)	DED	RT	CCUCON[31]	CCUCON.LCK
	inst_scu/inst_ccu/fsm_state_s[1]	DED	RT	-	-
	inst_scu/inst_ccu/ccucon_valid_sync_s	NONE	-	-	-

Figure 2: Specification Table

D. Routine for determining reset values

After the reset domain of each register in the design has been automatically determined by structural analyses, a pseudo property is generated which activates a reset and yields a counterexample. From this, the reset values of all registers in that reset domain are extracted by way of a function available in the formal property checker which returns the bit-values of any signal from a given property counterexample- or witness trace at any timepoint. The check of this pseudo property is very fast and allows the extraction of the values for all signals and bits which have been reset due to the reset activation assumption in the property.

Although SFR-verification is a topic for regular functional verification, structural SFF-integration verification can already check whether reset values and domains of SFF-wrappers implementing SFRs are consistent to specifications.

In our SFF-insertion flow, these automatically determined reset values are used to configure the generic reset parameters of the corresponding SFF-wrappers.

V. Transforming Register Write Logic

A. Extracting synchronous assignments into combinational process

A new sub-routine which is not part of the SFF-integration-verification flow deals with the transformation of synchronous RTL processes with register write logic into combinational processes for computing the next state values. Some RTL designers follow a design style where they provide one combinational process for the next-state logic, and a second, synchronous process for assigning the next state value to the register signal. In this case, the combinational process can just be kept, and the synchronous process be removed, since the register update is instead implemented in

an SFF-wrapper. If a synchronous process writes several registers only some of which are to be protected, the process is split into one keeping registers not to be safeguarded and another combinational one for the write data values of those to be protected. For all combinational next state values and update vectors to be connected to the SFF-wrapper input ports, new signal declarations are inserted.

To handle synchronous signal assignments in which the combinational logic has not already been written into a separate process by the designer, a VHDL parser is used to analyze the process. From this analysis, a new, asynchronous process is generated, in which the signal is assigned based on the same conditions as in the synchronous process with the only change being that the clock edge condition as well as the reset branch are removed. The information from this parsing step is used to modify the existing HDL code in such a way that any other signals from the transformed process remain unaffected unless they too are supposed to be secured in SFFs. This ensures not only that the signals to be secured are assigned correctly to an SFF, but also that there are no side effects on signals which may be assigned in the same process. The use of a parser in this step helps to guarantee the correctness of the SFF integration regardless of the language constructs used, as well as making the conversion algorithm less specific to the target HDL. This is particularly important with regard to the goal of correctness by construction as mentioned above, as well as the development of a SystemVerilog version of the tool.

The parser used to extract this information was generated using ANother Tool for Language Recognition (ANTLR4) [6]. This tool allows for the generation of a parser in a specified target language given a grammar in extended Backus-Naur form (EBNF). For this use case, ANTLR4 was used with the grammar of the VHDL93[7] standard to generate a VHDL parser implemented in Python[8]. Python was chosen as a target language due to its ease of use and relative speed compared to TCL[9], however, other targets such as C++ or Java would have also been possible. This made it comparatively simple to obtain a parse tree from any syntactically correct VHDL document. ANTLR4 can also be used to generate so-called visitor classes which traverse such a parse tree in a depth-first search and can be modified to perform specific actions when encountering certain language constructs (e.g. a VHDL signal assignment). Such a visitor was then used to extract from a synchronous VHDL process the information required to construct an equivalent, combinational process.

```
fsm_seq_p : PROCESS(clk_i, reset_n_i)
BEGIN  -- fsm_seq_p
 IF (reset_n_i = '0') THEN
  --SFFed:  fsm_state_s     <= IDLE;
  --SFFed:  ccucon_reg_o <= (OTHERS => '0');
  idle_s <= '1';
 ELSIF (clk_i'EVENT AND clk_i = '1') THEN
  CASE fsm_state_s IS
   WHEN IDLE =>
    idle_s <= '1';
    IF (ccucon_valid_sync = '1') THEN
    --SFFed  fsm_state_s    <= CAPTURE;
    END IF;
   WHEN CAPTURE =>
    idle_s <= '0';
    IF (reload_i = '1') THEN
     -- SFFed:  fsm_state_s    <= ACK;
      --SFFed  ccucon_reg_o  <= ccucon_i;
    END IF;
   WHEN ACK =>
    idle_s <= '0';
    IF (ccucon_valid_sync = '0') THEN
     --SFFed  fsm_state_s  <= IDLE;
      ...
    END IF;
   WHEN OTHERS =>
    idle_s <= '1';
    --SFFed:  fsm_state_s    <= IDLE;
  END CASE;
 END IF;
```

Figure 3: Reduced Synchronous VHDL Process

```
fsm_comb_p : PROCESS(ccucon_reg_o, fsm_state_s,
                reload_i, ccucon_i)
BEGIN  -- fsm_comb_p
  nx_ccucon_reg_o  <= ccucon_reg_o;
  upd_ccucon_reg_o <= (others => '0');
 nx_fsm_state_s <= fsm_state_s;
 upd_fsm_state_s <=  (others => '0');
 CASE fsm_state_s IS
   WHEN IDLE =>
     IF (ccucon_valid_sync = '1') THEN
       nx_fsm_state_s   <= CAPTURE;
       upd_fsm_state_s   <=  (others => '1');
     END IF;
   WHEN CAPTURE =>
    IF (reload_i = '1') THEN
      nx_fsm_state_s    <= ACK;
      upd_fsm_state_s <=  (others => '1');
     nx_ccucon_reg_o  <= ccucon_i;
     upd_ccucon_reg_o <= (others => '1');
     END IF;
  WHEN ACK =>
    IF (ccucon_valid_sync = '0') THEN
      nx_fsm_state_s  <= IDLE;
      upd_fsm_state_s <=  (others => '1');
    END IF;
 WHEN OTHERS =>
    nx_fsm_state_s    <= IDLE;
   upd_fsm_state_s   <=  (others => '1');
 END CASE;
END PROCESS fsm_comb_p;
```

Figure 4: New Combinational VHDL Process

The code example in Fig. 3 shows a synchronous process after transformation, where the registers *fsm_state_s* and *ccucon_reg_o* are hardened, while the register *idle_s* is kept unprotected. All assignments in the synchronous process which are moved to the new combinatorial process shown in Fig. 4 are just commented out. The combinational process preserves the control structure of the clocked sub-block of the original synchronous process.

B. Type conversions

In above example, the protected register *ccucon_reg_o* already has bit-vector type, but for the state signal *fsm_state_s*, the result of the combinational process as shown in Fig. 4 needs to be converted from the original enumeration type into bit-vector type. For enumeration types, encodings are available, which can be directly used for generating conversion functions in both directions. Either default encodings have been defined by the RTL designer, or an *enum_attribute* has been added to introduce a user-defined encoding, which could be a one-hot encoding for safety-purpose. Especially in the latter case, it is questionable whether the one-hot encoding is still needed, as the register is now going to be protected by the SFF-mechanism. For now, we use the original encoding for conversion of the enumeration type into bit-vector type and backwards. The conversion functions are generated automatically by a sub-routine which reads the type declaration. In above example, let us assume the type declaration to look like:

```
TYPE fsm_t IS (IDLE, CAPTURE, ACK);
ATTRIBUTE enum_encoding OF fsm_t: TYPE IS "001 010 100";   (6)
```

Figure 5

The generated VHDL conversion functions directly follow this encoding.

```
FUNCTION conv_fsm_t_sulv(ARG : fsm_type)
RETURN std_ulogic_vector IS
  BEGIN
   CASE ARG is
    WHEN IDLE =>
      RETURN "001";
    WHEN CAPTURE =>
      RETURN "010";
    WHEN ACK =>
      RETURN "100";
    WHEN OTHERS =>
      RETURN "000";
  END CASE;
  END conv_fsm_t_sulv;
```

Figure 6: Conversion to Bit-Vector

```
FUNCTION conv_sulv_fsm_t(ARG:
std_ulogic_vector) RETURN fsm_type IS
  BEGIN
   CASE ARG is
    WHEN "001" =>
      RETURN IDLE;
    WHEN "010" =>
      RETURN CAPTURE;
    WHEN "100" =>
      RETURN ACK;
    WHEN OTHERS =>
      RETURN IDLE;
  END CASE;
  END conv_sulv_fsm_t;
```

Figure 7: Conversion to State Type

With the standard encoding, the VHDL attribute *'pos* can just be used for this purpose.

```
inst_sff_wrapper_wrp0: sff_wrapper
 GENERIC MAP ( sff_pmeth_g => sff_pded_c,...)
 PORT MAP (
   clk_i     => clk_i,  reset_n_i => reset_n_i, ...,
   sff_d_i   => nxv_wrp0_s,  sff_ud_i  => upd_wrp0_s,sff_d_o   => wrp0_s,sffar_o   => sffar_o);
nxv_wrp0_s      <= conv_fsm_t_sulv(nx_fsm_state_s) & nx_ccucon_reg_o;
upd_wrp0_s      <= upd_fsm_state_s & upd_ccucon_reg_o;
ccucon_reg_o    <=  wrp0_s(ccucon_reg_o'range);
fsm_state_reg_o <=  conv_sulv_state_t(wrp0_s(nxv_wrp0_s'left downto ccucon_reg_o'length);
```

Figure 8: Instantiation of SFF-Wrapper

C. *Instantiation of SFF-wrapper*

The converted and concatenated next-state and update vectors are connected to the corresponding SFF-wrapper inputs, and the output is converted back and assigned to the original signal which had been the unprotected register signal before. In the example in Fig. 8, we assume the two previously mentioned signals to be protected together, and the protection method is configured as "double-error-detection", as assumed to be chosen in said specification table.

VI. Automatic Register Hardening Flow

The complete top-level procedure takes the compiled RTL design and a specification table with the registers to be safeguarded. Internally, it comprises four major steps:

1. Collecting data needed for optimum pre-structuring of the set of all register bits to be protected into SFF-wrappers depending on clock- and reset domains, instance hierarchies, protection methods, self-testability requirements, and maximum wrapper widths. If ECC (error correcting code) protection[10] is specified as protection method, i.e. double-bit error detection and optionally additional single-bit error correction, combining more bits into one wrapper is advantageous, as the required ECC-width grows just logarithmically with the data width. On the other hand, depending on the maximum clock frequency in which the specific module shall be operated, the allowable combinatorial run-time for ECC-computation is limited, therefore the user can specify the maximum SFF-wrapper width. By this preparatory step, an implementation plan is generated which contains the logical wrapper partitioning of register bits to be protected and more information to place and route the SFF-components to be inserted in the next steps.
2. Replacing the registers with SFF-wrappers, connecting the data-inputs and -outputs of it as described.
3. Inserting alarm reductors and connecting their alarm input vectors to the alarm outputs of the SFF-wrappers which belong to the same SFF-domain.
4. Inserting the SFF-umbrella controller and configuring it according to the total number of SFF-domains, connecting the domain alarm inputs to the corresponding alarm reductors or SFF-wrappers for each domain and in the other direction the wiring of the domain test-vectors to the SFF-wrappers of the same domain, and the global connections to the safety management unit.

Since the automatic SFF integration flow requires connecting automatically generated components as well as instances and signals that do not yet exist, a customized wiring algorithm was developed to complete the specific wiring tasks required for the SFF integration. This has the additional advantage that the SFF instantiation, process modification and signal wiring can be done in one step without the need for recompilation, speeding up the integration flow. To achieve this, the same parser that is used for converting synchronous signal assignments to be asynchronous above is reused to connect the new instances required to integrate the SFF components into a design.

Dedicated formal properties are additionally generated with wrapper architectures instantiating original and safeguarded design components to prove that mission function is not affected by the design transformation. They are structured like usual induction proofs with a base case in which the equivalence of corresponding signal bits is checked after reset, and a step case checking preservation of equivalence from one cycle to the next.

If a module design already contains SFF, it can occur that some previously safeguarded registers are decided not to be protected in a specific other product for a different use case in non-safety applications, so that redundant area can be saved. As the SFF-wrappers can be configured to implement individual bits just as regular flip-flops, the approach included in the automatic flow is to just adapt the configuration parameters accordingly but keep the SFF-wrappers in the module. If all bits of resulting SFF-wrappers are configured as unprotected, the alarm outputs of just left unconnected.

In summary, the automatic flow generates an implementation plan based on user specification and design analysis, transforms register assignment processes, inserts and configures the SFF-library components bottom-up, and wires these up through the instance hierarchy and to the module interface.

VII. Results and Discussion

We have shown the applicability of the flow for system control units from Infineon 32-bit microcontroller families with several 10k lines of code as a pilot, where the complete code transformation takes < 10 minutes including

identification of clock domains, reset classes and values, once the original unprotected design has been elaborated in the formal-property-checking environment so that all structural analyses can be run.

As in our current product family the register-monitoring mechanism has already been installed manually, the benefit for productive usage of the automated flow will be fully visible for future families. Nevertheless, there may well be changes for new derivatives of the current product family, where the automatic flow will be applied not only for modification of the protected register set, but also for optimization of safety architectures with respect to area consumption and unnecessary signal complexities which also affect the legibility of manually generated RTL code.

In total, we see following major benefits of the new flow:

1. Design effort reduction:
   Up to now, a design engineer has needed several days using the manual methodology to familiarize with the register-monitoring concept, and then further days to weeks to integrate the safety mechanism, all on top of the regular design effort for the mission function. With the automatic flow, this extra effort is saved.
2. Chip area
   The automated flow generates an optimal result due to structural analyses a designer not having deeper knowledge about the safety mechanisms would typically not be able to achieve. In particular, the partitioning of register flip-flops into SFF-wrappers has an impact on the total amount of extra flip-flops and logic gates which can be better controlled by the automatic flow.
3. Verification effort reduction:
   Despite our claim to yield correctness by construction, we would not recommend skipping integration verification, but we expect at least saved efforts for verification, bug analyses and corrections.

We plan to address SystemVerilog[11] designs and optimization of previously hand-written safeguarded designs. As the development was example-driven by representative modules, we might have to augment the procedure for designs with rare HDL features not yet supported. In the worst case, the designer could still update the automatically generated safety enhancements, but spending much less effort than adding the safety components completely manually like in the past. Even if our presented methodology is based on a company-proprietary safety library, the general approach could be applied to other libraries of pre-defined components which encapsulate configurable function and are supplied together with well-defined and machine-readable integration rules, possibly by third-party IP vendors.

Overall, we conclude that the step from automatic verification to design generation is feasible and very beneficial. Such automation is simplified by a uniform design concept based on well-defined library components, which allow all relevant signals and constants to be identified and filtered. The resulting RTL design is handled like previously fully hand-written RTL designs, where physical separation rules according to ISO26262 are followed in later steps. Like any automatic procedure, the real benefit depends on how often it is applied, and on the effort saved each time. In our chip projects with many safeguarded ASIL D modules and different product derivatives, it is advantageous.

We believe that our approach, which is based on expert knowledge and experience with this safety mechanism in combination with automated design analysis, is (still?) superior to generative AI which would have to be fed with a big collection of potentially suboptimal manually transformed designs by individual designers with varying understanding of the methodology who may not fully grasp all complex signal dependencies within the design.

ACKNOWLEDGMENT

Our thanks go to colleagues for discussions and valuable inputs, and management for support and encouragement.

REFERENCES

[1] ISO 26262 "*Road vehicles – Functional safety*", 2011, revised in 2018.
[2] H. Busch, *Integration Verification of Safety Components in Automotive Chip Modules*, Proceedings of DVCon-EU 2023.
[3] J. Ross, *Automatic Hardening of Registers in Safety Critical Modules*, Bachelor Thesis at Technical University of Munich / Infineon, 2023
[4] *TCL language*, https://wiki.tcl-lang.org/page/What+is+Tcl
[5] *OneSpin formal verification solutions*, https://eda.sw.siemens.com/en-US/ic/questa/onespin-formal-verification
[6] *ANTLR parser generator*, https://www.antlr.org/index.html
[7] *IEEE Standard VHDL Language Reference Manual*, in ANSI/IEEE Std 1076-1993 , vol., no., pp.1-288, 6 June 1994, doi: 10.1109/IEEESTD.1994.121433.
[8] Python Software Foundation, *Python Language Reference, version 3*, http://www.python.org
[9] *Programming language benchmark by GitHub user Kostya*. https://github.com/kostya/benchmarks
[10] M.Y. Hsiao, *A Class of Optimal Minimum Odd-weight-column SECDED Codes*, IBM Journal of R & D Vol. 14, July 1970, pp. 395-401
[11] S. Sutherland et al., *SystemVerilog for Design Second Edition*, Springer New York, NY, 2006.