

# Automated Formal Verification of Area-Optimized Safety Registers in Automotive SoCs

Shuhang Zhang\*, Bryan Olmos\*<sup>†</sup>,

\*Infineon Technologies AG

<sup>†</sup>Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau

Email: [Firstname.Lastname@infineon.com](mailto:Firstname.Lastname@infineon.com)

## Abstract

Registers are primary storage elements in System-on-chip (SoC) designs and play an important role in maintaining state information and processing data in digital systems. With respect to the ISO26262 standard, these registers require high levels of reliability and fault tolerance. For this reason, safety-critical applications require that normal registers are equipped with additional safety components to construct safety registers, which ensure system stability and fault tolerance. However, the process of integrating these safety registers is complex and error-prone, because of highly-configurable features provided by a safety library such as parameterized modules and flexible safety structures. In addition, to address the overhead caused by the safety registers, we have applied area optimization techniques to their implementation. However, this optimization can make the integration process more susceptible to errors. To avoid any integration mistakes, rigorous verification is always required, but it is time-consuming and error-prone if the verification is implemented manually when dealing with numerous verification requests. To address these challenges, we propose an automated flow for the verification of safety registers with the formal approach. The results indicate that this automated verification approach has the potential to reduce the verification effort by more than 80%. Additionally, it ensures a comprehensive examination of every requirement of this safety library, which is reflected in faster detection of bugs. The proposed framework can be replicated for the verification of other safety components enabling an early detection of potential issues and saving valuable time and resources.

## I. INTRODUCTION

In the modern System-on-Chip (SoC) design, registers hold a paramount position, because these fundamental components not only act as the primary storage units within digital systems but also maintain state information. For automotive applications, registers are also required to operate under strict safety and reliability standards, e.g. ISO26262 [1], which necessitates a high degree of consistency and fault tolerance. To meet these stringent requirements, the ordinary registers are usually armed with safety components to ensure that automotive chips can tackle faults effectively and maintain their operation without compromising the safety and reliability of the overall system. To implement these safety features in registers, a specific safety library, named Safety Flip Flop (SFF) library [2], is developed and utilized, providing different safety components to implement various safety features within the register implementation.

However, the integration of these safety registers is also an error-prone process due to the flexibility provided by the SFF library. Firstly, these safety registers themselves are parameterized and highly configurable. For example, this library supports different types of protection methods for registers, from Error Correction Code (ECC) to Redundancy, and provides optional self-test features. In addition to safety registers, the safety library provides two additional types of safety components: alarms and controllers. Designers have the flexibility to arrange and combine these components to create various safety structures. However, this configurability, stemming from parameterized modules and flexible safety structures, poses significant challenges in verification. Furthermore, safety designs often lead to substantial area overhead, which is cost-inefficient. To mitigate this, area-optimization techniques are typically employed. Despite their benefits, the implementation of area-optimized registers introduces additional challenges during the integration process. Moreover, manually-implemented verification is time-consuming, especially when dealing with numerous verification requests, because safety registers are utilized in almost all blocks of SoC designs. Therefore, a more efficient approach is needed in our case.

Recognizing these challenges, we propose an automated workflow for the verification of safety components in register implementations. This approach aims to replace the traditionally manual verification process, significantly reducing the time and effort required while simultaneously eliminating the potential for human error. By introducing

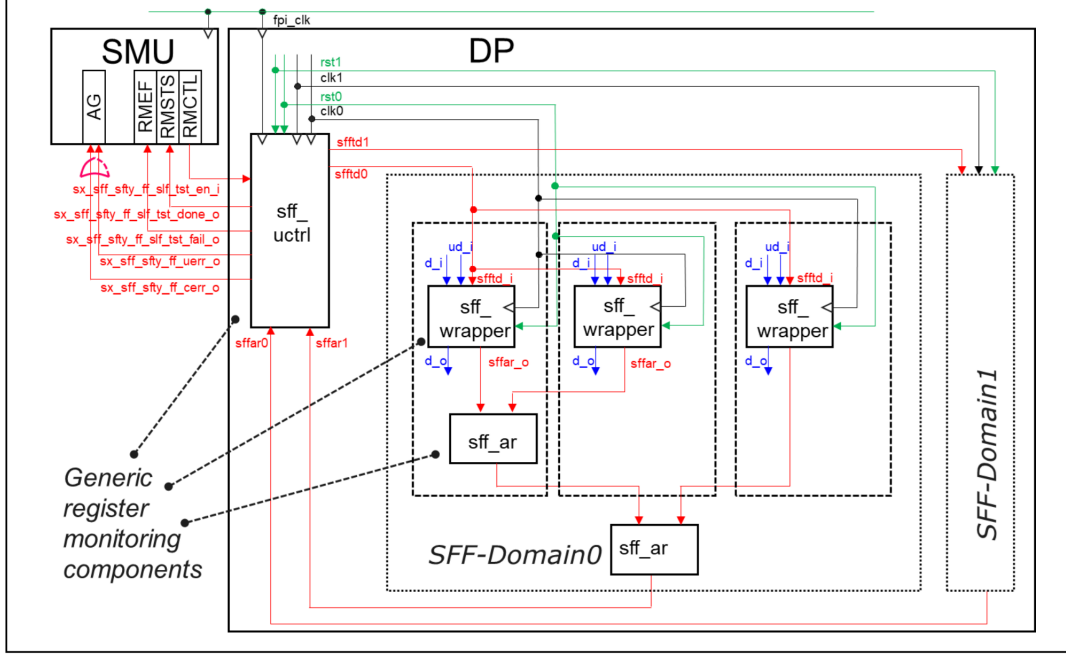


Fig. 1: An example of a safety structure with safety components provided by the SFF library. [3]

automation, we can dramatically increase both the speed and accuracy of the verification process. More specifically, the contributions of this work include:

- We analyze the safety library and develop a verification plan to cover all requirements of safety components from the SFF library, as well as area-optimized implementations. Additionally, we demonstrate all defined property classes in this work.
- We propose a novel verification flow to generate the integration properties based on safety specifications and defined property classes automatically. These properties can be verified by commercial verification tools naturally.
- Our results show that the manual effort for the safety verification can be reduced significantly, from around 5Person-Day (5PD) to 1PD for one safety structure. In the last project, our formal framework detected three integration bugs and two additional bugs in area-optimized implementations. In addition, a comparison with a state-of-the-art work [3], which focuses on SFF integration verification, is also discussed from different aspects.

The rest of this paper is organized as follows. In Section II, we introduce the SFF library and highlight the motivation for the automated formal verification framework. The proposed formal verification framework will be elaborated in Section III, and the verification results will be presented in Section IV. At last, we conclude this work in Section V.

## II. BACKGROUND AND MOTIVATION

In this section, we present some background knowledge regarding the SFF library, including area-optimized approaches and basic concepts related to the proposed framework. We also highlight the challenges associated with the verification of safety components.

### A. Safety Flip Flop (SFF) Library

A detailed diagram of the SFF library is presented in Fig. 1. This library provides three main components for constructing safety register implementations in automotive chips: the SFF Wrapper, the Alarm, and the Controller. These components are explained in detail as follows:

- **SFF Wrapper:** This is the most important component provided by the library, replacing ordinary register implementations with safety registers. The library supports five different protection methods, classified into two categories: Error Detection and Correction Code (EDCECC) [4], and Redundancy [5]. The specific methods include Parity, Double Error Detection (DED), Single Error Correction Double Error Detection (SECEDED),

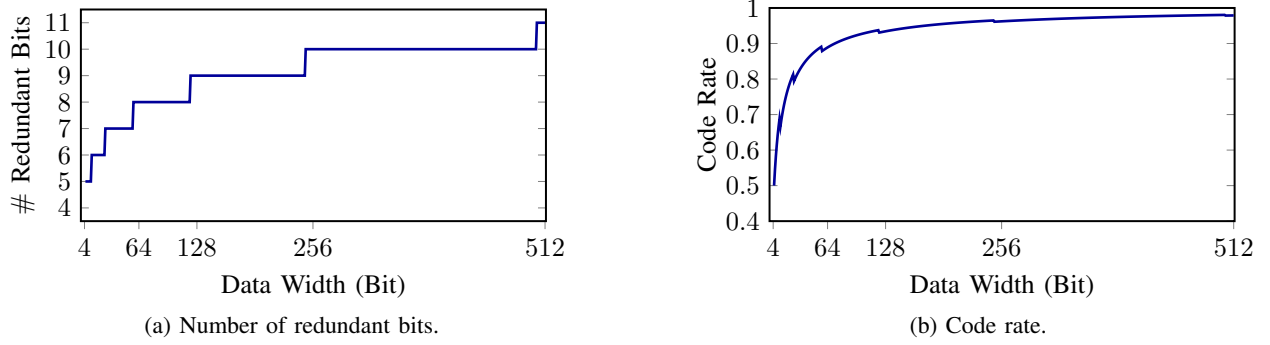


Fig. 2: The number of redundant bits (a) and code rate (b) in a SECDED code implementation with the data width ranging from 4 to 512.

Double Module Redundancy (DMR), and Triple Module Redundancy (TMR). Additionally, optional self-test features [6] are supported for the safety registers.

- **SFF Alarm:** This component is used to collect alarm signals from multiple safety wrappers or from different clock domains and generate a reduced alarm signal vector. These reduced alarm signals will be sent to upper-level alarm instances or the SFF Controller.
- **SFF Controller:** This component communicates with the upper-level Safety Management Unit (SMU). If an error is indicated by the reduced alarm signals, an error flag is sent to the SMU. Additionally, this component controls the self-test features. It receives a trigger signal from the SMU and generates test sequences for all safety registers in different clock domains.

### B. Area-Optimized Safety Implementation

1) *Motivation:* The EDCECC technique is typically employed in safety registers to provide fault-tolerance features, but it also introduces redundant data bits. Given that safety registers consume substantial area, designers usually aim to protect only the critical bits to save area. As a result, a large number of small safety registers are instantiated, which is highly area-inefficient. To illustrate this area inefficiency, we use the SECDED code as an example. The number of redundant bits is shown in Fig. 2(a), and the code rate, defined as  $\frac{\#DataBits}{\#TotalBits}$ , is shown in Fig. 2(b). These figures clearly indicate that larger safety registers are more area-efficient. To mitigate the area overhead caused by these small registers, merging them into larger ones is a commonly adopted strategy.

2) *Optimization Strategies:* This section explains two optimization approaches based on different algorithms. The first approach addresses solving a bin-packing problem using the Best-Fit Decreasing (BFD) technique [7], which groups small safety registers only considering their size. For example, the original design is shown in Fig. 3(a) with three 32-bit registers, labelled as *a*, *b*, and *c*, instantiated within the register block. However, only part of these registers are protected by safety registers: Register *a* and *b* have two bits protected each (sff\_wrapper\_1 and sff\_wrapper\_2), while Register *c* has 30 bits protected (sff\_wrapper\_3). We would like to emphasize that in the original design, each register is limited to having only one safety register, and this safety register cannot contain data bits from other registers. Meanwhile, the remaining unprotected bits are not displayed in this figure. In Fig. 3(b), the optimization results of the BFD algorithm are presented, where the original safety registers are replaced with one large and one small register, considering only the size information of all safety registers. Here, sff\_wrapper\_1, with a data width of 32 bits, includes the data bits from both Register *c* and *a*. Meanwhile, sff\_wrapper\_2 retains a width of two bits and includes the data bits for Register *b*.

The second approach uses the Integer Linear Programming (ILP) technique [8], including in the analysis not only the size information of all safety registers but also their synthesized area data. Fig. 3(c) showcases the ILP-optimized design. In this case the algorithm solution is the use of two medium-sized safety registers. In this scenario, sff\_wrapper\_1 has a data width of 17 bits, protecting part of Register *c*'s data bits. The other safety register contains the remaining bits from Register *c*, along with data bits from Register *a* and *b*. Although the ILP-based approach requires synthesis data during optimization, this is a one-time effort for the same technology, which does not introduce significant additional effort. Consequently, the ILP-based approach can break down large registers and achieve greater area reduction than the BFD-based approach. Both optimized designs achieve greater area efficiency compared to the original design.

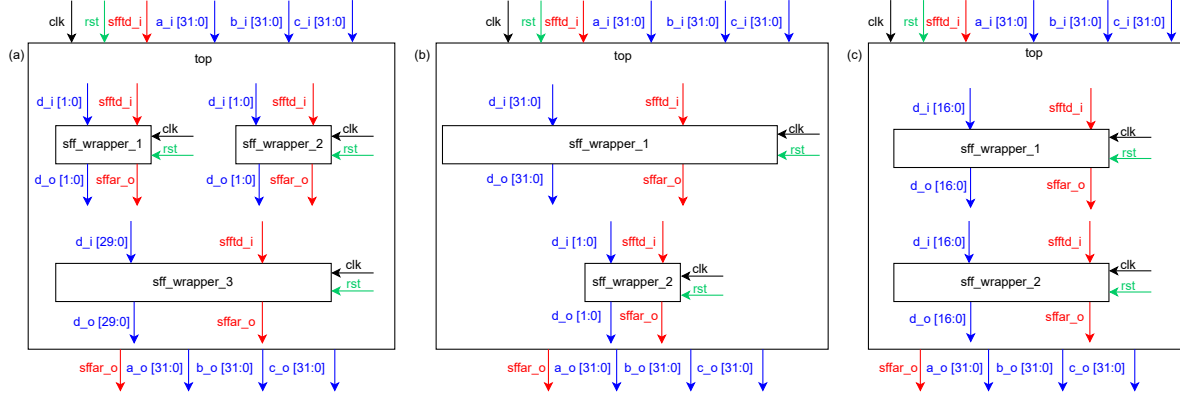


Fig. 3: (a) Original design. (b) Optimized design with algorithm 1. (c) Optimized design with algorithm 2.

### C. Verification Challenges

As mentioned in the previous sections, the verification challenges are caused by the highly-configurable features. These features can be classified into two parts:

- **Parameterized Components and Configurable Safety Structure:** Each component from this safety library is configurable, and these components can be combined flexibly to construct various safety architectures to meet different safety requirements. While this flexibility is highly convenient for the design team, it introduces significant challenges for the verification team. To manage the high level of configurability, the formal verification environment must be adapted based on the configurations of the safety architectures being used.
- **Area-Optimized Implementations:** As discussed in the previous section, area-optimized safety registers are necessary to reduce manufacturing costs. Multiple safety registers are merged to form larger ones, thereby reducing area consumption. However, as a result of the merging process, the one-to-one correspondence between signals and registers no longer exists. Additionally, the merging results vary depending on the chosen algorithm, complicating the determination of which signal should connect to the respective parts of the merged registers. Furthermore, the merging is executed automatically by an algorithm, making it essential to verify that the merging adheres to specifications.

To address these issues in the safety register verification and achieve exhaustive results, we have proposed an automated formal verification framework.

## III. FRAMEWORK

As mentioned in previous sections, the highly-configurable nature of the safety library, along with the area optimization techniques, makes the verification of safety registers a challenging task. Therefore, we primarily address this task with our proposed verification framework, illustrated in Fig. 4. The proposed verification flow is depicted in Fig. 4(a), while the property generation flow is shown in Fig. 4(b). The property generation flow consists of three steps: Specification Extraction, Property Definition, and Property Generation. In this framework, the verification process is broken down into several steps, each considering different abstraction models. A detailed introduction is provided for each step in this section.

### A. Specification Extraction

The specifications of safety registers are defined in an XML file, and all safety register information will be extracted from this file based on a pre-defined model, as shown in Fig. 5, which is comparable to the technique utilized in our register generator [9]. The extracted information is stored in a specification instance, named the Model of Things (MoT), for subsequent steps. Additionally, some extra options are used for the property generator. For example, the area optimization algorithm is defined in the options. Since different optimization algorithms generate different safety register instances, it is crucial to consider the chosen algorithm during property generation. Therefore, the optimization results, which specify which safety registers are merged into which groups, is also part of the specification instances. It is important to highlight that the same optimization algorithm must be implemented twice by both the design and verification teams, as the specification only includes the algorithm type. During this

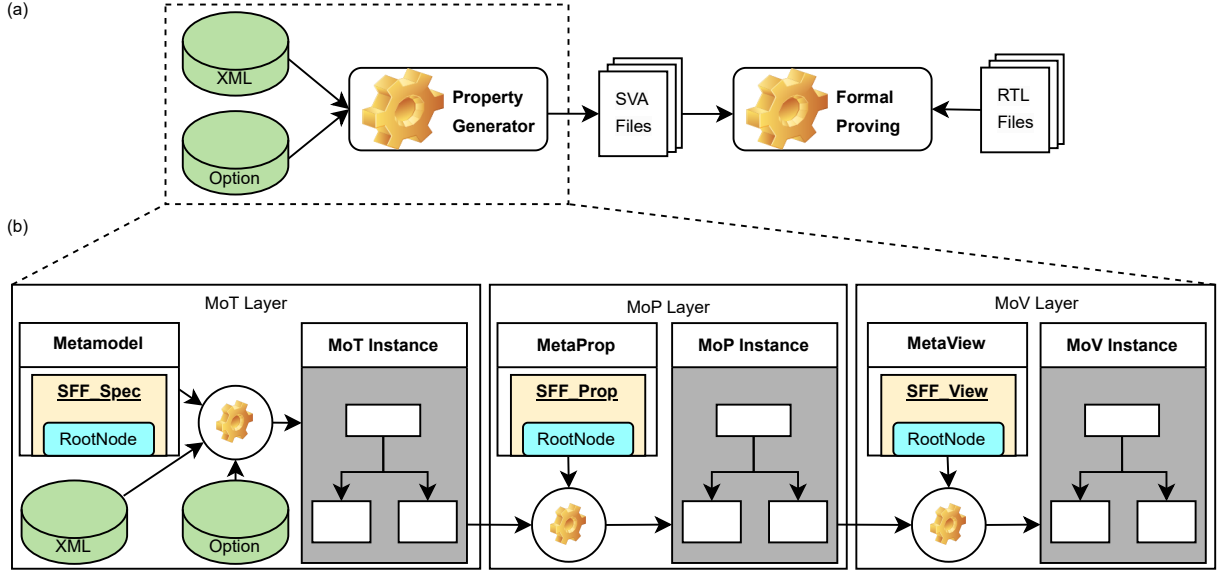


Fig. 4: Proposed property generation flow for safety registers.

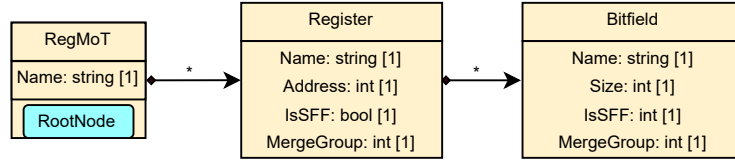


Fig. 5: Simplified model for specification extraction.

phase, our framework supports the checking of the optimization results between the algorithm implementations of both the design and verification teams. This capability can be used to detect algorithm bugs at a very early stage.

### B. Property Definition

1) *Utilized Property Classes*: As area optimization results in completely different safety register instantiations when compared with the initial designs, the generated properties should also differ. In this section, we will discuss multiple property classes in both area-optimized and non-area-optimized modes in the Model of Properties (MoP). The property classes utilized in this work include the following:

- **Configurable Parameters**: This property checks if the instantiation of safety components is correct, ensuring that parameters such as data width and protection method match the specifications.
- **Safety Mode**: This class evaluates mainly the behavior of safety components under fault injection conditions. For safety registers, an error signal should be asserted when faults are injected. The number of faults is determined by the protection method.
- **Connectivity**: This property class verifies the connectivity of safety components. For example, it focuses on the input and output connectivity of safety registers with primary inputs and outputs.
- **Normal Mode**: This class is used to check the basic functionality of safety components, such as the basic register functionality.
- **Self-Test Mode**: This class checks the self-test features of safety registers. During this phase, fault injection is also considered.

Due to page limitations, we cannot present all the properties used in our projects. Therefore, we will focus on three classes—Configurable Parameters, Safety Mode, and Connectivity—that are most relevant to area-optimized designs to demonstrate the property differences between original and optimized designs.

2) *Comparison between Original and Area-Optimized Designs*: The example properties of the Parameters, Safety Mode, and Connectivity property classes are illustrated in Fig. 6 and Fig. 7 for the example designs in

```

property param_dw_safety_registers;
1 |->
sff_wrapper_1.dw == 2 &&
sff_wrapper_2.dw == 2 &&
sff_wrapper_3.dw == 30;
endproperty

```

(a) Parameter Checking - Original Design

```

property param_dw_safety_registers;
1 |->
sff_wrapper_1.dw == 32 &&
sff_wrapper_2.dw == 2;;
endproperty

```

(c) Parameter Checking - Optimized Design 1

```

property param_dw_safety_registers;
1 |->
sff_wrapper_1.dw == 17 &&
sff_wrapper_2.dw == 17;
endproperty

```

(e) Parameter Checking - Optimized Design 2

```

property fault_injection_safety_registers;
// Num. fault patterns decided by its data width
// Three properties generated
'Fault_Injection_i |->
sff_wrapper_i.err_o == 1;
endproperty

```

(b) Fault Injection Checking - Original Design

```

property fault_injection_safety_registers;
// Two properties generated
'Fault_Injection_i |->
sff_wrapper_i.err_o == 1;
endproperty

```

(d) Fault Injection Checking - Optimized Design 1

```

property fault_injection_safety_registers;
// Two properties generated
'Fault_Injection_i |->
sff_wrapper_i.err_o == 1;
endproperty

```

(f) Fault Injection Checking - Optimized Design 2

Fig. 6: Comparison of connect properties for original and optimized designs.

Fig. 3. In Fig. 6, the parameter and fault-injection properties are shown. Fig. 6(a), (c), and (e) demonstrate the properties for parameter checking. In the original design, there are three registers instantiated, so all the data width parameters must be checked. However, with area optimization techniques, the number of safety registers is reduced, resulting in different generated properties based on the utilized algorithm. Since our property generator also implements the optimization algorithm internally, these merged data width values can be determined, thus updating the properties automatically. Additionally, in Fig. 6(b), (d), and (f), the safety mode properties, specifically fault-injection properties, are demonstrated for both original and area-optimized designs. Similar to parameter checking, the original design requires three properties to cover the fault tolerance features due to the instantiation of three safety registers, whereas area-optimized designs require only two properties. Although the number of properties for area-optimized designs is reduced in fault-injection mode, the proving time for each property increases. This is because a merged safety register implies a larger data width, resulting in a significant increase in the number of error states considered during the verification phase. The number of error states to be considered for EDCECC-based implementations can be formalized as follows:

$$N = 2^{N_d} \cdot \sum_{i=1}^{N_e} \binom{N_d + N_r}{i}, \quad (1)$$

where  $N_d$ ,  $N_r$  and  $N_e$  represent the number of data bits, redundancy bits and the number of supported errors, respectively. Based on this equation, it is evident that a larger data width leads to a significant increase in the state space. If the size of the merged register is too large, the state explosion will result in an unbounded proof, which should be avoided. Therefore, in our project, we introduce a constraint on the maximum allowed merging size from the verification perspective. The verification time for both the original and optimized designs will be discussed in detail in the results section.

Apart from the parameter and fault injection checking, the connectivity check is also critical for the optimized designs. The input and output connectivity checking for the original design is presented in Fig. 7(a) and (b). However, when the smaller registers are merged, the one-to-one mapping relationship between the safety registers and the original registers disappears. Consequently, the connectivity checking properties must be updated based on the optimization algorithms, making verification challenging even for small designs. Our proposed verification framework can automatically generate the adapted properties without requiring any manual effort. The adapted properties for the optimized designs can be found in Fig. 7(c) and (d) for input connectivity checking and in Fig. 7(d) and (f) for output connectivity checking.

```
property conn_top_input_safety_registers;
1 |->
top.a_i[1:0] == sff_wrapper_1.d_i[1:0] &&
top.b_i[1:0] == sff_wrapper_2.d_i[1:0] &&
top.c_i[29:0] == sff_wrapper_3.d_i[29:0];
endproperty
```

(a) Input Connectivity Checking - Original Design

```
property conn_top_input_safety_registers_optimized_1;
1 |->
top.c_i[29:0] == inst_wrapper_1.d_i[29:0] &&
top.a_i[1:0] == inst_wrapper_1.d_i[31:30] &&
top.b_i[1:0] == inst_wrapper_2.d_i[1:0];
endproperty
```

(c) Input Connectivity Checking - Optimized Design 1

```
property conn_top_input_safety_registers_optimized_2;
1 |->
top.c_i[16:0] == inst_wrapper_1.d_i[16:0] &&
top.c_i[29:17] == inst_wrapper_2.d_i[12:0] &&
top.a_i[1:0] == inst_wrapper_2.d_i[14:13] &&
top.b_i[1:0] == inst_wrapper_2.d_i[16:15];
endproperty
```

(e) Input Connectivity Checking - Optimized Design 2

```
property conn_top_output_safety_registers;
1 |->
sff_wrapper_1.d_o[1:0] == top.a_o[1:0] &&
sff_wrapper_2.d_o[1:0] == top.b_o[1:0] &&
sff_wrapper_3.d_o[29:0] == top.c_o[29:0];
endproperty
```

(b) Output Connectivity Checking - Original Design

```
property conn_top_output_safety_registers_optimized_1;
1 |->
sff_wrapper_1.d_o[29:0] == top.c_o[29:0] &&
sff_wrapper_1.d_o[31:30] == top.a_o[1:0] &&
sff_wrapper_2.d_o[1:0] == top.b_o[1:0];
endproperty
```

(d) Output Connectivity Checking - Optimized Design 1

```
property conn_top_output_safety_registers_optimized_2;
1 |->
sff_wrapper_1.d_o[16:0] == top.c_o[16:0] &&
sff_wrapper_2.d_o[12:0] == top.c_o[29:17] &&
sff_wrapper_2.d_o[14:13] == top.a_o[1:0] &&
sff_wrapper_2.d_o[16:15] == top.b_o[1:0];
endproperty
```

(f) Output Connectivity Checking - Optimized Design 2

Fig. 7: Comparison of connect properties for original and optimized designs.

	Initial Design #SafetyReg	Verification Time Overhead	
		BFD (32-bit)	ILP (32-bit)
SoC1	1827	6.53%	14.87%
SoC2	2392	71.05%	84.32%
SoC3	49	144.27%	131.02%

TABLE I: Comparison of verification time.

### C. Property Generation

After defining all property classes, the properties can be generated based on different view requirements in the Model of View (MoV). In this work, we mainly focus on the use of System Verilog Assertions (SVAs). With these assertion files and the design files, the formal tool can be triggered to obtain the verification results.

## IV. RESULTS

### A. Setup

In this work, we applied our proposed framework using safety register data from three automotive SoCs. This property generation framework is based on Python, with the area optimization algorithms embedded in the same generator. The ILP solver used in this work is PuLP [10]. The generated properties are verified using a commercial property checking tool.

### B. Formal Verification Results

1) *Verification Time:* Although the same property classes are defined for both the original and optimized designs, the generated properties differ based on the chosen algorithm, which results in varying verification times across different designs. In particular, the safety mode properties can consume significantly more time in the optimized designs compared to the original designs due to the larger data width of the merged registers. In Table I, we present the overhead in verification time (property proving time) for three SoCs with the maximum allowed register size set to 32 during the optimization. In this work, all properties can still converge because of the constraints provided by the verification team. From this table, it is evident that the verification time overhead varies across different SoCs and different algorithms. Based on our observations, if small-sized safety registers, such as 4-bit or 8-bit, comprise a large portion of the design, the verification time overhead is significant. This is because these registers will be merged into a 32-bit safety register, and the safety-related properties consume significantly more time, as seen in the case of SoC3.

	Area-Opt.	CEX Trace	Functional	Integration
This Work	yes	yes	yes	yes
Work [3]	no	no	no	yes

TABLE II: Comparison between This Work and Previous Work [3].

2) *Detected Bugs*: This proposed framework has been applied to a production project. Using it, we identified five bugs related to safety integration—three in connectivity and two in the area-optimized implementation—in a short period. The bugs found in the area-optimized designs were also related to connectivity. One bug we identified was that some bits of merged registers were not connected under specific configurations. Another bug was related to the clock and reset connections in the optimized designs. This is because optimized designs obscure the one-to-one mapping between register specifications in the XML file and the registers instantiated in the RTL code. Additionally, since all properties are generated automatically, the verification effort for each safety structure in this project has been reduced from five person-days (PD) to one PD.

### C. Comparison with Another Verification Framework

After discussing the verification results with our proposed framework, we would like to compare it with previous work that also focuses on safety register verification. The comparison items are listed in Table II. In the previous work, area-optimized designs are not supported, and it still relies on a one-to-one mapping between specification and RTL code to conduct the verification. Additionally, the previous work extracts design information from RTL to construct multiple tables, allowing connectivity checking to be shifted to table checking without triggering any formal tools. Although this approach is faster, it makes debugging difficult if the check fails and a Counter-Example (CEX) trace is required. Moreover, the previous work covers safety register verification only partly, as it splits the verification into two phases: functional and integration. In the functional phase, a simplified safety architecture is verified for its main safety features, such as fault injection features, by an individual team. Then, integration verification is conducted based on the generated tables. In contrast, our work verifies all features with the complete register blocks. Both functional and integration aspects are verified simultaneously, ensuring comprehensive coverage and reducing the risk of overlooking any potential issues.

## V. CONCLUSION

Safety registers are critical components in modern automotive SoCs, requiring extensive verification to ensure correct behavior. Safety features often introduce significant area overhead, making area-optimized approaches essential for more efficient automotive SoCs. However, optimizing for area introduces additional verification challenges, as optimized designs may not naturally adhere to the initial specifications. To address these verification challenges, we propose an automated formal verification framework. This framework automatically generates properties based on register specifications and the chosen optimization algorithm. These properties can be seamlessly used with commercial formal verification tools. The framework has been applied to a production project, significantly reducing human effort and enabling the discovery of bugs in area-optimized designs within a short time period. Lastly, we would like to highlight that this verification approach is not limited to our in-house SFF library. The verification flow can be adapted for other safety libraries as well, providing a versatile and scalable solution for safety register verification across different projects and platforms.

## ACKNOWLEDGMENT

We also want to thank our colleagues, Basavaraj Naik, Vidya Sagar Kantamneni, Yifang Wang, Xin An and Djones Lettnin, for their valuable input and suggestions.

## REFERENCES

- [1] "ISO26262 Road Vehicles – Functional Safety, Part 1-10," International Organization for Standardization (ISO), Standard, 2018.
- [2] H. Busch, "Automated Safety Verification for Automotive Microcontrollers," in *Design & Verification Conference U.S.*, 2016.
- [3] H. Busch, "Integration Verification of Safety Components in Automotive Chip Modules," in *Design & Verification Conference Europe*, 2023.
- [4] R. W. Hamming, "Error detecting and error correcting codes," *The Bell system technical journal*, vol. 29, no. 2, pp. 147–160, 1950.
- [5] F. L. Kastensmidt, L. Sterpone, L. Carro, and M. S. Reorda, "On the optimal design of triple modular redundancy logic for sram-based fpgas," in *Design, Automation and Test in Europe*, 2005, pp. 1290–1295.
- [6] E. J. McCluskey, "Built-in self-test techniques," *IEEE Design & Test of Computers*, vol. 2, no. 2, pp. 21–28, 1985.
- [7] C. Kenyon, "Best-fit bin-packing with random order," Ph.D. dissertation, Laboratoire de l'informatique du parallélisme, 1995.
- [8] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
- [9] S. Zhang, B. Olmos, and B. Naik, "Automated Formal Verification of a Highly-Configurable Register Generator," in *Design & Verification Conference U.S.*, 2024.
- [10] PuLP, "Pulp: A python linear programming library," <https://pythonhosted.org/PuLP/>, accessed: 2024-10-01.