Special Session: AutoSoC

A Suite of Open-Source Automotive SoC Benchmarks

Augusto da Silva, Felipe; Bagbaba, Ahmet Cagri; Ruospo, Annachiara ; Mariani, Riccardo ; Kanawati, Ghani ; Reorda, Matteo Sonza; Jenihhin, Maksim; Hamdioui, Said; Sauer, Christian

DOI
10.1109/VTS48691.2020.9107599

Publication date
2020

Document Version
Accepted author manuscript

Published in
2020 IEEE 38th VLSI Test Symposium (VTS)

Special Session: AutoSoC - A Suite of Open-Source Automotive SoC Benchmarks

Felipe Augusto da Silva*†, Ahmet Cagri Bagbaba*‖, Annachiara Ruospo‡, Riccardo Mariani¶, Ghani Kanawati§, Ernesto Sanchez‡, Matteo Sonza Reorda‡, Maksim Jenihhin‖, Said Hamdioui† and Christian Sauer*

*Cadence Design Systems
Munich, Germany

†Delft University of Technology
Delft, The Netherlands

‡Politecnico di Torino
Turin, Italy

§ARM Ltd
Austin, TX, USA

¶Nvidia Corporation
Milan, Italy

‖Tallinn University of Technology
Tallinn, Estonia

***Abstract*—The current demands for autonomous driving generated momentum for an increase in research in the different technologies required for these applications. Nonetheless, the limited access to representative designs and industrial methodologies poses a challenge to the research community. Considering this scenario, there is a high demand for an open-source solution that could support development of research targeting automotive applications. This paper presents the current status of AutoSoC, an automotive SoC benchmark suite that includes hardware and software elements and is entirely open-source. The objective is to provide researchers with an industrial-grade automotive SoC that includes all essential components, is fully customizable, and enables analysis of functional safety solutions and automotive SoC configurations. This paper describes the available configurations of the benchmark including an initial assessment for ASIL B to D configurations.**

Keywords - Automotive benchmark; SoC; open-source; Functional Safety; ISO 26262.

I. INTRODUCTION

In recent years, advances in technology enabled the employment of automated systems to control driving tasks. The idea of electronic devices having full control over a vehicle promises to change the concept of mobility in the near future. However, allowing computers to control all the tasks in a vehicle requires high complexity systems and major concerns with respect to the safety. The development of Autonomous Vehicles applications, where a system failure could cause life-threatening situations, entails in state-of-the-art challenges on different aspects of system development. Concerns with Reliability, Security, Quality, and compliance to Safety Standards are of high priority. This scenario requires adoption of new techniques and methodologies that will facilitate development and verification of these applications. Several organizations are working to close the technological gap for Autonomous Vehicles. However, in order to assess the quality of the proposed solutions, it is necessary to compare the results against what is applied in the industry. Nowadays, development lifecycles and verification techniques applied by industry are not disclosed, and each big player in the automotive sector has its own methodologies and tools. In addition, there is limited access to automotive hardware and software solutions. This is a challenge for researchers, that may not be able to verify their work in representative designs or assess the quality of their results. For that reason, there is a high demand for a suite of open-source benchmarks that would enable research on the different aspects of Automotive applications development. It should be outlined that the benchmarks should include not only the hardware description (at different levels of abstraction), but also compatible software modules (Operating System, peripheral drivers, sample applications) and information about the implemented safety and security mechanisms.

As part of the efforts for developing solutions to address the demands of Autonomous Driving, industry and academia are investing in research on several related areas. Several works are exploring aspects of fault-tolerance in hardware architectures [1], [2], software design [3], operational systems [4], among others. [5] provides a broader look on specific reliability challenges for autonomous systems, for both automotive and robotics. The challenges of Functional Safety compliance, based on standards like ISO 26262, are also explored in research as [6], [7]. The authors point out Fault Injection (FI) Simulation as one of the critical steps for compliance with the standard. For that reason, different approaches are proposed to leverage FI Simulation, optimization of the simulation techniques [8], [9], combination of multiple fault analysis technologies [10], analysis of faults on different hardware abstractions levels [11], [12], and many others. Several works are also discussing the security issues imposed by these applications [13]. Challenges with hardware attacks [14] and secure in-vehicle communication [15] are being investigated and their interference with functional safety and reliability is getting to the front. Although several works include significant contributions to advance the state-of-the-art, they all have some common pitfalls. First, experiments are usually not performed on representative designs. Results may be compromised by a lack of comprehensive test cases, which should be based on Systems on Chip (SoCs) with an operating system and software applications that are representative of the Automotive sector. Also, such systems should be fully open-source, allowing different researchers to assess the quality of

the results by comparison. Even though some components of such systems are available in the community, to the best of our knowledge no open-source package including SoC hardware models, OS and SW applications, that is representative of the Automotive sector is available.

To address these challenges, we propose an open-source industrial-grade benchmark suite. The proposed Automotive benchmark comprises all its elements in the format of an SoC, and hence, it was named AutoSoC. The AutoSoC was conceived by the analysis of commercial solutions, and considering common development techniques deployed by industry. The selected architecture considered the availability of software (compilers, debuggers, operating systems, and others) and the feasibility of development in multiple hardware abstraction levels (Virtual Platform, RT and gate level). The suite includes multiple configurations with different levels of Safety Mechanisms (SMs), enabling investigation of Functional Safety aspects. The AutoSoC appears as an interesting candidate to support Automotive research. The main contributions of our work are:

- Launch the initiative for an open-source SoC benchmark suite for Automotive applications
- Provide a solution for integrating inter-layer components and their interoperability required for an automotive SoC development
- Demonstrate representative use cases by a set of software applications including an Automotive Cruise Control
- Validate the concept by including a preliminary Safety Assessment targeting different ASIL configurations.

The AutoSoC benchmark suite is available for download in *http://www.autosoc.org*.

The remainder of this paper is organized as follows. Section II elaborates on the reasons behind the need for standardization and benchmarking in the automotive as well as in the closely related robotics domain. Next, Section III describes the definition of the functional requirements for the AutoSoC based on the characterization of industrial solutions. Afterwards, in Sections IV, V and VI, we describe its base HW and SW components, the Safety components and the available benchmark configurations. Section VII outlines a preliminary functional safety analysis targeting different ASIL configurations. Last, Section VIII presents our conclusions and future work.

II. SAFETY STANDARDIZATION AND BENCHMARKING FOR AUTOMOTIVE AND ROBOTICS

Nowadays, highly automated safety-critical systems (such as autonomous vehicles and autonomous mobile robots) are implemented with very complex integrated circuits. They are composed of a large set of HW elements, executing an equally large set of SW elements, often from third parties. This complexity has created a strong demand for standardization initiatives related to semiconductors, to guarantee uniformity, interoperability and repeatability of the many activities required by a safety lifecycle. The main initiative is the 2nd edition of ISO 26262, with a part 11 [16] fully dedicated to the application of ISO 26262 to semiconductor technologies. The part 11, with its 179 pages, provides a detailed set of guidelines on principles, methods and architectures for digital, mixed signal, programmable device and sensor type of integrated technologies. The variety of solutions and combinations provided by part 11 is huge, as also the opportunity to create new ideas fulfilling the principles highlighted by the standard.

On the other hand, that vastity of options is a challenge from several points of view. For example, despite the ISO 26262 provides a mathematical approach to quantify the probability of failure due to HW random failures, it is very effort intensive to apply it and quickly compare the effectiveness of each proposed solution. In fact, the results are highly dependent on the chip architecture and the related SW application executed on it. The same challenge exists for the verification activities (e.g. fault injection) required to confirm the effectiveness of some of the functional safety properties, such as the diagnostic coverage. The time spent to setup each fault injection campaign for each different architecture solution makes unpractical to use it during the exploration phase – so limiting the creativity and the space of possible solutions. Another challenge is caused by the interaction between several different properties and requirements. For example, a typical approach to achieve high diagnostic coverage is the so-called loosely coupled lock-step, i.e. the same SW is executed redundantly in two different processing cores and compared by a third element. The resulting diagnostic coverage highly depends on how the SW redundancy is executed (e.g. if it is a task per task or instruction per instruction redundancy, if the OS is in common or shared, etc.), on how often the two SW executions are compared, on how many variables of the compared SW are exposed to the comparison, etc. It is also necessary to evaluate the so-called Diagnostic Time Interval, i.e. how often it is possible to perform that comparison and the time required by the Safety Mechanisms to compare and detect the potential failure. As also it is necessary to evaluate the degradation of performance (e.g. in terms of worst case execution time or WCET) that the comparisons of the loosely coupled lock-step are causing to the data traffic of the nominal functionality.

The complexity described by the previous examples indicates the strong need of an open-source benchmarking environment, to provide scientists with a ready-to-use and clearly defined platform on which to implement and test safety solutions in a comparable way. That platform, for example, should allow researchers to compare two different implementations of the loosely coupled lock-step scheme. Another use case for that benchmarking environment is the measure of the application overhead caused by the execution of SW test libraries (STLs), a well-known method described in ISO 26262 part 11. The availability of a common benchmark will allow a transparent and well defined comparison of the impact to the application caused by two different STL implementations.

III. AUTOMOTIVE SOC ARCHITECTURES

This section describes the analysis of commercial automotive SoCs that led to the definition of the functional blocks of

the AutoSoC. The gathering of requirements for the proposed SoC considered the main features available in well-known automotive solutions. The objective of this characterization was to create an SoC that is representative of the industry standards.

A. Industry Solutions Characterization

Nowadays, the industry is embedding several features in SoCs targeting different in-vehicle applications. The so-called Automotive Ecosystem includes solutions for infotainment, powertrains, network communication, automatization of driving tasks, among others. All those features require robust solutions that must consider aspects of functional safety and security. Although different commercial solutions are available, in general, architectures have similarities that can be explored to define a set of requirements for an Automotive SoC. The requirements for the AutoSoC were gathered based on an analysis of the datasheets of commercial Automotive SoCs. We considered the main characteristics of available solutions to identify common aspects that can be regarded as mandatory by the industry. In general, the analysis can be split into the following domains:

1) Hardware Architecture: common architecture characteristics;
2) Safety: what components of the SoCs are considered for functional safety compliance and which safety mechanisms are usually implemented;
3) Security: which security features are available;
4) Other: commonly available peripherals (e.g. communication protocols, GPUs, Audio/Video DSPs).

One notable common characteristic, among the evaluated solutions, is the availability of multiple CPUs. In general, dedicated hardware components are available for safety-critical and application-specific operation. This concept allows the deployment of powerful CPUs for applications with high processing demands (e.g. video processing), while safety-critical applications are executed in CPUs with dedicated safety mechanisms. For example, the Renesas R-Car M3 [17] includes two CPUs for common applications and an additional Dual Lockstep CPU for safety-critical applications. The Infineon AURIX [18] and Texas Instruments TDA2SG [19], follow a similar concept by including a CPU and separated cores for dedicated functionalities. Dual-Core Lockstep (DCLS) is the most common safety mechanism available for CPUs. For the memories, including RAMs and caches, industrial solutions usually deploy Error Correction Codes (ECCs) and Parity. DCLS, ECCs, and Parity have an advantage regarding Functional Safety analysis. These SMs are introduced by the recommendations of ISO 26262 [20] and include a reference of their fault coverage capabilities. Hence, by deploying any of these SMs as described in ISO 26262, the referenced Diagnostic Coverage can be directly used during Functional Safety Analysis.

The other components available in the analyzed SoCs could be categorized as communication protocols, application-specific, security, and infrastructure peripherals. In general, the commercial solutions implement a good variety of communication peripherals, including automotive protocols as CAN and FlexRay, and general protocols as Ethernet, SPI, and I2C. Another common characteristic is the availability of Video and Audio dedicated hardware. As the majority of the SoCs aim to Advanced Driver-Assistance Systems (ADAS) applications, they include peripherals like GPUs, video codecs, Image Processing Units, and Audio DSPs. In the security domain, apart from proprietary features that are not detailed, the most common components are cryptography engines, like Advanced Encryption Standard (AES), Data Encryption Standard (DES), Hash, among others. Also, some solutions provide access control features like firewalls and protected memory areas. Additionally, every analyzed solution included infrastructure peripherals like JTAG, UART, GPIO and debug components.

TABLE I
SUMMARY OF COMMERCIAL SOC ANALYSIS

	Renesas R-Car M3	**Infineon AURIX**	**Texas TDA**
Safety CPU with DCLS	+	+	-
Memories with ECC	+	+	+
Second CPU (no SM)	+	+	+
Dedicated Video IPs	+	+	+
Automotive Peripherals	+	+	+
Security Cripto IPs	+	-	+

Considering the characteristics of the evaluated commercial solutions, it is possible to define a common set of features that can be seen as required by the automotive industry. The addition of safety-related components, application-specific units, automotive protocols, and security cores, can be established as the basic set of features for a representative Automotive SoC. The summary of common characteristics found in the evaluated commercial solutions is available in Table I.

B. AutoSoC Functional Blocks

Based on the characterization of industrial solutions, summarized in Table I, an initial architecture of AutoSoC was established. Functional blocks were defined aiming to cover the minimum set of features required for a representative automotive benchmark suite. The concept of functional blocks is also important to keep the design modular. Different versions of AutoSoC can deploy diverse hardware components to cover the requirements of each functional block. Figure 1 illustrates the outcome of our analysis.

As it happens in most commercial solutions, the AutoSoC has two main processing units. The *Safety Island* is responsible for all safety-critical processing capabilities. It is composed of CPUs and memories that must be covered by Safety Mechanisms according to the requirements of ISO 26262. The division between safety-related hardware and the rest of the SoC components supports the compliance with Functional Safety standards, as only the safety-related hardware is required to comply with ISO 26262. The other processing unit is the *Application Specific Block*. This unit implements the

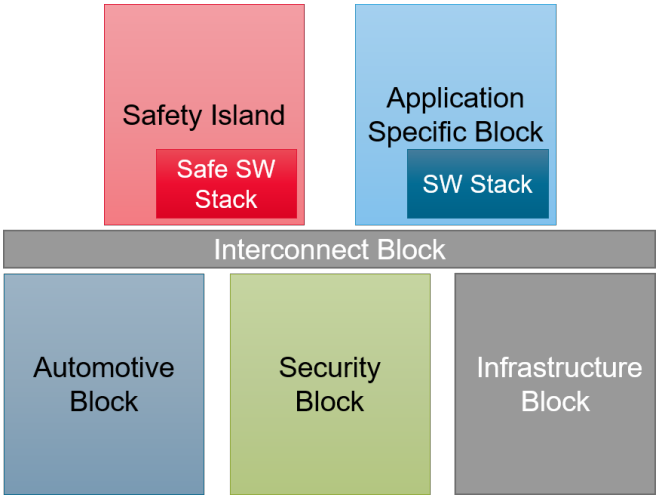

Fig. 1. AutoSoC Functional Blocks.

hardware required for application-specific processing. It may include CPUs and memories for high demand applications, GPUs and Image processing units for video applications, among others. The target functionality for each given AutoSoC configuration will define the Hardware components required for the *Application Specific Block*. Also, it is important to notice that the *Safety Island* and the *Application Specific Blocks* have dedicated Software stacks. Both can execute distinct Operational Systems and applications that will better suit their requirements.

The remaining blocks implement communication, security, and general SoC infrastructure. The *Automotive Block* is responsible for SoC communication with in-vehicle systems. The most common protocol deployed for in-vehicle communication is CAN. However, other options can be implemented, like FlexRay, LIN, Automotive Ethernet, among others. The *Security Block* is responsible to perform all security-related functionalities of the AutoSoC. The most common employment is cryptography cores, like AES and DES. However, we expect other security features to be explored. With this, the AutoSoC benchmark architecture allows future extensions aiming at support the new security standard under development ISO 21434. The latter aims at defining a Cybersecurity Assurance Level (CAL), similar to the ASIL concept [21]. The *Infrastructure Block* is responsible for the on-line health monitoring of the SoC. It includes debugging features such as JTAG and UARTs to ease the development process. Finally, the *Interconnect Block* is responsible for internal SoC communication. It may deploy common communication buses, like AXI and Wishbone, or more advanced options such as a Network-on-Chip (NoC).

IV. AUTOSOC BASE COMPONENTS

This section outlines the processing units, interconnect components, debug elements, and software workloads currently integrated into the AutoSoC. An initial configuration of the benchmark, named AutoSoC QM, is set up by deploying only the base components. The AutoSoC QM is a fully functional version of the benchmark and works as the foundation for further configurations. The modular design of the AutoSoC allows additional configurations to be instantiated by simply enabling additional Safety components. The next sections describe the available Safety components and AutoSoC configurations.

A. Hardware Components

The selection of the CPU, as the central unit of the AutoSoC, considered different processor architectures, performance features (e.g. pipeline stages and memory interfaces), main buses, software stacks, and the possibility of development on multiple abstraction levels (Virtual Platforms, RT level, and gate level). A further requirement is that the CPU has to be open-source. Different analyzed options could be considered as good candidates for the CPU. For instance, the Amber2 [22] is a 32-bit RISC CPU compatible with the ARM v2a instructions set. Another considered option was the Gaisler LEON3 [23]. It includes a 7 stages pipeline, a comprehensive set of peripherals, and support scripts. This work has deployed the OpenRISC [24] (mor1kx implementation) as the main CPU. The OpenRISC includes a better variety of support tools, an active community and the resources for the development of a Virtual Platform. Also, the community supports a variety of compatible peripherals that can be easily integrated, including CAN, AES, and DES [25].

The OpenRISC community provides tools and examples for the development of SoCs. As part of that, there is an example SoC based on the mor1kx CPU. The package includes CPU, memory, UART, JTAG, and a debug unit, all connected with a Wishbone bus. Also, the example SoC contains a testbench with features for loading software applications to the memory and connection to the debug unit via JTAG. This example was used as a base for the AutoSoC. By deploying the example, we can cover the infrastructure and interconnect blocks. Also, we can reuse part of the provided test environment to speed up the development.

B. Software Resources

One of the objectives of the Automotive Functional Safety analysis is to avoid disturbance of the safety-related functionalities of a system by random hardware fault. In the case of an SoC, the software application executed by the CPU defines the functionality. For that reason, the software stack is an important part of the Functional Safety analysis. The current version of AutoSoC includes several software options. The intention was to integrate the available resources and the applications developed by ourselves in a unified repository in the AutoSoC simulation environment. The simulation of all available software applications is possible by suitably setting up the configuration files. AutoSoC includes several software resources organized by folders. The Baremetal folder includes development resources as Makefiles, drivers, and around 50 compiled test applications. Also, a compiled Linux kernel (bootable in simulation) is available in the Linux folder.

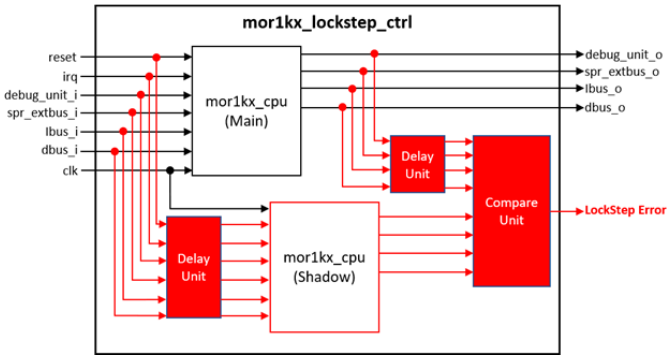

Fig. 2. Time diversity Dual-Core Lockstep implementation.

Furthermore, the RTEMS folder includes a development environment with Makefiles, drivers, and applications. Finally, an *Automotive Cruise Control* application was developed. The application is based on the RTEMS Operational System. It comprises four real-time tasks for reading vehicle sensor data, computing actuation, setting some engine parameters, and housekeeping.

V. AUTOSOC SAFETY COMPONENTS

Another important aspect of the benchmark is the availability of Safety Mechanisms in the Safety Island. As this block is responsible for executing safety-critical applications, we need to assure that potential faults can be detected avoiding possible harm to the expected functionalities. The CPU, as the primary unit of the Safety Island, is the primary target for the safety evaluation. Different safety mechanisms schemes were conceived, each targeting different Automotive Safety Integrity Levels (ASIL).

A. Dual-Core LockStep

The first option deploys time diversity Dual-Core Lockstep (DCLS) as the main Safety Mechanism. The DCLS configuration includes a redundant copy of the CPU, delay units for time diversity and compare units for fault detection. The implementation of the DCLS with time diversity is illustrated in Figure 2.

The performance of the main processor is not affected by the DCLS implementation. The main CPU is the only one with write access to the bus, controlling the functionality of the SoC. On the other hand, the shadow CPU does not perform any write access to the SoC resources. Instead, the outputs of the shadow CPU are used only by the Compare Unit for fault detection. In case of a mismatch between the outputs of both processors, an alarm is activated by the Compare Unit. Despite the additional fault coverage by including DCLS, we still need to consider the effect of common-mode failures that can impact both processors and are not detectable by comparison of the their outputs [26]. To minimize the potential of common-mode failures the DCLS mechanism includes time diversity. Time diversity works by applying a delay in the execution of the shadow processor. The delay is obtained by including a delay unit in the driven signals of the CPU. Delay units are also added to the outputs of the main processor, to align both core outputs for the Compare Unit. The Delay Units can be configured with the desired time shift: the current version applies a delay of 2 clock cycles to all signals. The shadow CPU execution delay configuration must consider the system requirements for maximum fault tolerance time. Since this delay is also applied to the input of the Compare Unit, a mismatch between the CPU outputs will be detected only after the configured delay.

Dual-Core Lockstep is the most used SM scheme for processors targeting ASIL D applications. However, not all applications demand ASIL D and the extra cost of including a redundant copy of the CPU. For that reason, AutoSoC incorporates additional configurations targeting different ASIL requirements.

B. Software Test Libraries

A Software Test Library, also referred to as STL, is a collection of software tests that are run on power-on (key-on), power-off (key-off) or periodically to prevent faults from leading to single-point failures or prevent them from becoming latent as a result of a multiple-point fault.

This software mechanism aims at detecting permanent faults that can occur anytime during the execution of a safety application and can cause a safety violation. An STL corresponds to a set of software procedures, usually developed in assembly code, C code or a combination of both. These may be executed either at boot-time or run-time. In the former case they require supervisor capabilities and therefore, to avoid conflict with the Operating System (OS), are usually executed during the power-on and power-off. On the other hand, when the STLs are executed at run-time, they have to coexist with the OS. Then, it is essential to make these tests run in a short period of time, usually few milliseconds, to avoid affecting the behavior of the other software applications running on the same hardware. The software scheduler will schedule these tests at specified time intervals when the hardware is idle or running less time sensitive applications.

In the recent years, several semiconductor and IP companies started to provide their customers with Software Test Libraries (STLs) to be used for on-line fault detection when the target devices are used in safety-critical applications. The advantage stemming from their adoption lies first of all in the fact that system companies can test their products in the field while guaranteeing a given fault coverage, even without knowing the implementation details (black-box testing). Moreover, STLs perform the test exactly in the system operating conditions, thus executing at speed and avoiding any overtesting. Finally, they do not require any change in the hardware, thus avoiding any area or performance overhead. On the other side, the generation of STLs is mainly manual at the moment and requires special skills in order to achieve sufficiently high fault coverage figures. Computing these figures for a given STL also requires a new generation of tools called Functional Fault Simulators. Several recent works introduced guidelines on how to correctly generate STLs for CPUs [27], [28] and

peripherals [29], how to speed up the FI experiments [30], how to maximize their fault coverage in the different scenarios (possibly minimizing the test time [31]), and how to re-use existing STLs.

C. Internal Memories ECC

Usually, in complex CPUs internal memories occupy the highest area on the physical device. As the component size is directly related to the probability of faults, the internal memories are a primary target for SMs. The ISO 26262 standard includes recommendations for well-known memory Safety Mechanisms. Based on the recommendations and the findings of the industry solutions characterization, Error-Detection-Correction Codes (ECC) was selected as an option to protect the internal memories of the CPU. The current implementation of the Safety Island CPU includes seven blocks of internal RAMs. Together, the internal memories represent 91.3% of the total fault targets in the RT level representation of the CPU. The deployment of an SM with high Diagnostic Coverage, like ECC, on all internal memories, will provide a satisfying coverage for the overall CPU.

D. External Memory ECC

The other elements of the Safety Island must also be verified for the possibility of single points of failure. Generally, software applications must be loaded to the external memory to be executed by the CPU. Also, the applications utilize the memory for storing data and control parameters. As the software application function relays on the external RAM, memory failures have a direct impact on the intended functionality. The external RAM must also be covered by ECC to avoid propagation of internal memory faults to the outputs of the Safety Island.

E. Bus Parity

The data bus is responsible for data transmission between the memory and the CPU. For that reason, a fault in the data bus could propagate to the CPU or to the memory and would not the detected by their SM. To avoid these cases, a parity checker was included to cover data transmissions between CPU and memory. The Parity checker monitors data bus transmissions, and calculates a Parity bit for all communications between CPU and memory. The Parity bit is transmitted by a direct connection between the Parity Check blocks. In case of a wrong parity, an alarm is set to inform the system.

F. Checkpoint Control

Even if the DCLS SM is employed, both CPUs could get stuck in the same software instruction, and none of the mentioned SMs would be able to detect this fault. For that reason, a Checkpoint Control safety mechanism was implemented. The Checkpoint control monitors the Data Bus expecting predetermined software signatures in specific memory locations. The mechanism works as a Hardware Watchdog, but instead of expecting a single refresh from the software application, it expects a different signature for each software task. Consequently, the SM is capable of verifying not only if the software application is running, but also if the Control Flow is as expected. The Checkpoint Control is fully customizable during elaboration, allowing the definition of the software signatures, expected sequence, and deadlines.

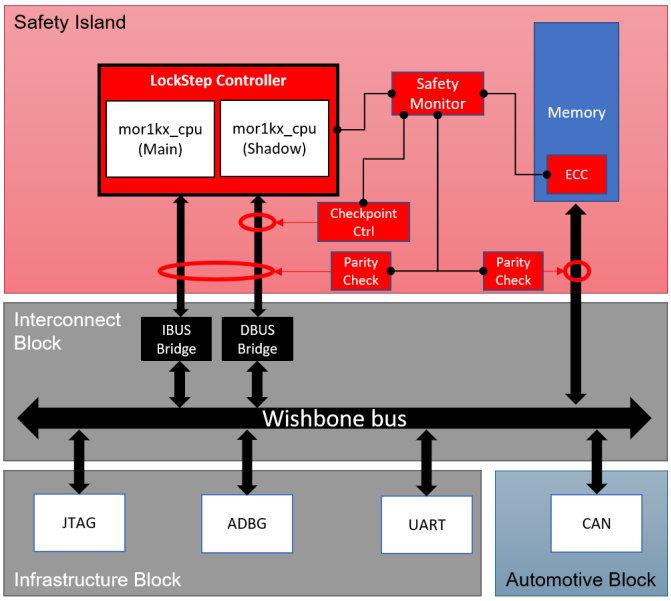

Fig. 3. AutoSoC Safe Configuration.

G. Safety Monitor

Finally, a Safety Monitor block was developed to integrate all the detection alarms. In the case of fault detection of any SM, the Safety Monitor generates an external alarm and an error code to indicate where the fault was detected. Figure 3 illustrates the architecture of the AutoSoC Safe configuration, including the DCLS, External Memory ECC, Bus Parity and Checkpoint Control.

VI. AutoSoC Configurations

This section outlines the available benchmark configurations and how they can be set up by enabling the different safety components. The available configurations comply with the Functional Blocks: Safe, Automotive, Infrastructure and Interconnect. The Application Specific and Security Blocks, as illustrated in Figure 1, will be developed in the next stages of our work. The modular design of the AutoSoC allows the reuse of the Functional Safety Analysis, performed in the scope of this paper, on later configurations.

As part of its modular concept, several configurations of the AutoSoC are possible by enabling different combinations of the mentioned components. For defining a new configuration, based on the provided simulation folder, the user must select the Hardware components in the elaboration config file, choose the software application and enable any combination of Safety Mechanisms by adding defines to the 'plus args' config file

TABLE II
AUTOSOC CONFIGURATIONS

Benchmark Configurations	Dual Core LockStep	Internal Mem ECC	Software Test Libraries	BUS Parity	Checkpoint Control	Safety Monitor
AutoSoC QM	-	-	-	-	-	-
AutoSoC ECC	-	+	-	-	-	-
AutoSoC STL	-	+	+	-	-	-
AutoSoC DCLS	+	-	-	-	-	+
AutoSoC SAFE	+	-	-	+	+	+

(e.g. +define+DCLS). The new configuration can then be elaborated and simulated with the provided Makefile. Although any possible combination of components can be created, we have defined a group of initial configurations for the AutoSoC. These configurations are based on common SM combinations from industry solutions. Table II illustrates some potential configurations for the AutoSoC. For the scope of this paper, we have performed a preliminary safety assessment for three configurations. The configurations AutoSoC ECC, AutoSoC STL, and AutoSoC DCLS, were analyzed as candidates to target different ASIL levels.

VII. PRELIMINARY FUNCTIONAL SAFETY ANALYSIS

This section describes the functional safety analysis of some of the available configurations of AutoSoC. Functional Safety Analysis, as specified by ISO 26262, aims to decrease the risk of failures caused by malfunctions. Within electronic systems, it focuses on avoiding that random hardware faults can disrupt the expected functionality of a design. The Automotive Safety Integrity Level (ASIL), defines the required risk reduction for a particular functionality. Functionalities with a higher risk of hazard situations demand a higher ASIL. In general, to reduce the risk of malfunctions induced by random faults, we include Safety Mechanisms (SMs). The required percentage of detection, or Diagnostic Coverage (DC), is defined by the ASIL.

Typically, Functional Safety analysis is completed at later stages of the hardware design. Additional parameters like area, Failure-in-Time (FIT) rate, and Failure Modes distribution, are necessary to confirm design compliance to the required ASIL. These parameters are used to calculate Safety Metrics that show the design capacity to cope with different fault models. For that reason, the current AutoSoC analysis is considered preliminary. The next step of our work is to finalize the gate-level description of AutoSoC, determine the possible failure modes, define the diagnostic coverage based on the failure mode distribution, and calculate the final safety metrics.

A. AutoSoC DCLS configuration

Hardware redundancy schemes, like Dual-core Lockstep, are defined by ISO 26262 as recommended safety mechanisms for processing units. The standard defines the typical diagnostic coverage for these mechanisms is high, meaning 99% of detection for random hardware faults. The implementation of DCLS should aim to provide early detection of failures, by step-by-step comparison of results produced by two processing units operating in lockstep. The AutoSoC DCLS configuration intends to comply with the description from ISO 26262. Also, the implementation of time diversity increases the DCLS features by addressing the effects of common-mode failures.

TABLE III
DCLS CPU FAULT COVERAGE

Fault Target	SA(1/0) Faults	Detected by DCLS	Residual Faults
mor1kx_cpu	675,504	668,749	6,755

A preliminary investigation of the mor1kx_cpu description shows a potential of 337,752 possible fault targets. If we consider the SA0 and SA1 fault models, as required for ISO 26262 permanent faults analysis, there are a total of 675,504 faults to be analyzed. The DCLS safety mechanism intends to identify faults in the mor1kx_cpu. By respecting the Diagnostic Coverage defined by ISO 26262 for the DCLS, we can assume that 99% of the faults in the mor1kx_cpu(Main) will be detected by the Lockstep Controller. With 99% of fault coverage, we can expect the AutoSoC DCLS to be a good candidate to comply with ASIL D requirements. Table III illustrates the potential fault coverage for the AutoSoC DCLS configuration.

B. AutoSoC ECC configuration

As described for the Processing Units, ISO 26262 also includes recommendations of Safety Mechanisms for Volatile and Non-Volatile memories. One of the recommendations is the deployment of Memory monitoring using Error-Detection-Correction Codes (ECC). Traditionally, ECC algorithms can detect every one and two-bit failures, and some three or more bit failures in a word. The standard defines the typical diagnostic coverage for ECC is also 99% of detection for random hardware faults. Usually, on complex CPUs, internal memories, or caches, occupy the largest area on the physical devices. For that reason, they will have a high contribution to the design Failure-In-Time (FIT) rate. This contribution will appear in the Failure Modes (FM) distribution, with cache-related FMs requiring Safety Mechanisms to decrease the residual FIT. It is a common design practice to protect the cache memories with ECC or Parity. In the AutoSoC design, the internal memories represents a potential of 633,344 possible fault targets considering the SA0 and SA1 fault models. This number represents 93.7% of the total number of fault targets for the entire CPU. For that reason, the addition of

TABLE IV
INTERNAL MEMORIES ECC FAULT COVERAGE

Fault Target	SA(1/0) Faults	Detected by ECC	Residual Faults
Fetch instructions cache ram	262,144	259,523	2,621
Fetch instructions cache tag ram	20,992	20,782	210
Fetch instructions MMU ram	8,192	8,110	82
Load/Store data cache ram	262,144	259,523	2,621
Load/Store data cache tag ram	19,968	19,768	200
Load/Store data MMU ram	8,192	8,110	82
Load/Store store buffer	51,712	51,195	517
TOTAL	**633,344**	**627,011**	**6,333**

SM to the internal memories represent a good overall coverage for the CPU faults. The AutoSoC internal ECC configuration considers the incorporation of ECCs to all internal memories. Table IV demonstrates the fault coverage of the ECC for each internal memory block. The total number of faults covered by the ECCs, considering the 99% DC defined by ISO 26262, is 627,011 faults. This coverage represents a 92.8% Diagnostic Coverage of the entire CPU. These figures acknowledge the AutoSoC internal ECC configuration as a good candidate to comply with ASIL B requirements.

C. AutoSoC STL configuration

To avoid the hurdle of the extra hardware required by DCLS schemes, there is an increasing demand for software strategies for the on-line testing of automotive processors. This section describes the main characteristics of the software test libraries being developed to improve the AutoSoC CPU fault coverage and reports the preliminary results.

Preliminary results are gathered on two AutoSoC CPU modules: the Arithmetic Logic Unit (ALU) and the Load and Store Unit (LSU). The STL programs have been developed resorting to three of the most common strategies for Software-Based Self-Test (SBST) generation [32]: ATPG-based, deterministic and evolutionary-based [33]. The current STL comprises 16 test programs for a total of 64 KB. The AutoSoC STL Configuration targets the CPU (mor1kx_cpu), cleared of all the possible sources of non-determinism such as Instruction Cache and Data Cache. Indeed, when *evaluating* the test programs fault coverage, the exact stream of instructions entering the pipeline must be deterministic: these modules might lead to a fluctuating fault coverage and therefore should be deactivated for the fault grading process (which directly contributes to the ASIL process certification) [34]. This does not prevent the caches (or similar) from being used when the STL is integrated in the application software and deployed in field.

Starting from these considerations, permanent faults injection analyses have been carried out on a total of 42,160 faults target for the mor1kx_cpu at RT level, and a total of 60,672 permanent faults for the mor1kx_alu and the mor1kx_lsu units at gate level. If considering the mor1kx_alu and mor1kx_lsu at RT level, there are 4,938 fault targets. The Fault Injections experiments were performed at both the RT and gate level, mimicking the typical process used in practice, where RT level estimations are used as a proxy for gate level fault coverage estimation during the STL development process. A further investigation was performed in order to identify all the untestable and safe faults [35], revealing a non-negligible increase in the fault coverage of the two targeted modules. Once again, the identification of untestable and safe faults represents a common issue in practice, given that their number may often be non negligible. Table V sums up the gathered results showing the achieved fault coverage on the ALU and LSU modules, both at the RT and gate level. The achieved Fault Coverage (FC) considering the redundant and safe faults is reported as Testable Fault Coverage (TFC).

TABLE V
SELECTED CPU MODULES STL FAULT COVERAGE

CPU Modules	RT-Level		Gate Level	
	FC [%]	TFC [%]	FC [%]	TFC [%]
ALU + LSU	68.71	80.04	76.23	85.43

The deployment of software routines to identify permanent faults is shown to be effective in multiple units of a CPU [35]. Although it is not always possible to achieve ASIL D fault coverage requirements by deploying STLs, they are an appealing alternative when combined with other Safety Mechanisms. A common practice in the automotive industry is to combine STLs with ECC in the internal memories of the CPU. For instance, in [35] the authors achieved a permanent fault coverage of 84.4% by deploying an STL in an OpenRISC CPU similar to AutoSoC CPU. The AutoSoC CPU contains 42,160 targets for stuck-at-0 and stuck-at-1 faults, not considering the internal memories. If we consider the fault coverage from [35], the STL would be able to detect 35,583 faults. If we include the STL routines in the AutoSoC ECC ConfigurationVII-B, the combined Safety Mechanisms would detect 662,594 faults. As the total number of faults is 675,504, the combined detection rate represents a Diagnostic Coverage of 98%. This figure would allow the combination of the AutoSoC STL and ECC configurations to be a good candidate to comply with ASIL C requirements.

VIII. CONCLUSIONS

The development of Autonomous Vehicles is driving the industry to close the technological gap demanded by these applications. The research community is proposing solutions to address the concerns with safety, security, performance, among others. However, it may be hard to assess the quality of their results. In most cases, there is limited access to representative designs and comparison with industrial methodologies is very complicated. To address this matter, we present the AutoSoC benchmark suite. Our work intends to provide researchers with an SoC that is based on commercial solutions, includes all essential components, is highly customizable, and allows comparability between distinct methodologies and results. This paper outlines the current architecture options incorporated in the AutoSoC, including hardware components, software applications, operating systems, and safety mechanisms. Also, we describe a preliminary functional safety assessment target-

ing different ASIL configurations. Further works on AutoSoC may focus on new Safety Mechanisms or combinations of them, new techniques to automate the safety analysis (e.g., to better identify untestable and safe faults) and make it faster (e.g., speeding up functional fault simulation), and to evaluate cross-layer solutions to evaluate and increase the system dependability. We believe that the availability of this benchmark suite will allow researchers to develop new solutions and to quantitatively assess their effectiveness, thus contributing to the advancement of the state of the art in the area.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 722325.

REFERENCES

[1] J. Han, Y. Kwon, Y. C. P. Cho, and H.-J. Yoo, "A 1ghz fault tolerant processor with dynamic lockstep and self-recovering cache for ADAS SoC complying with ISO26262 in automotive electronics," in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, nov 2017.

[2] A. B. de Oliveira, G. S. Rodrigues, and F. L. Kastensmidt, "Analyzing lockstep dual-core ARM cortex-a9 soft error mitigation in freeRTOS applications," in *Proceedings of the 30th Symposium on Integrated Circuits and Systems Design Chip on the Sands - SBCCI 17*. ACM Press, 2017.

[3] A. Höller, N. Kajtazovic, T. Rauter, K. Römer, and C. Kreiner, "Evaluation of diverse compiling for software-fault detection," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE Conference Publications, 2015.

[4] G. Rodrigues, F. Rosa, A. de Oliveira, F. L. Kastensmidt, L. Ost, and R. Reis, "Analyzing the impact of fault tolerance methods in ARM processors under soft errors running linux and parallelization API," *IEEE Transactions on Nuclear Science*, pp. 1–1, 2017.

[5] M. Jenihhin, M. Sonza Reorda, A. Balakrishnan, and D. Alexandrescu, "Challenges of reliability assessment and enhancement in autonomous systems," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, oct 2019.

[6] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, nov 2017.

[7] Y.-C. Chang, L.-R. Huang, H.-C. Liu, C.-J. Yang, and C.-T. Chiu, "Assessing automotive functional safety microprocessor with ISO 26262 hardware requirements," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*. IEEE, 2014.

[8] D. Alexandrescu, A. Evans, M. Glorieux, and I. Nofal, "EDA support for functional safety — How static and dynamic failure analysis can improve productivity in the assessment of functional safety," in *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE, jul 2017.

[9] A. Evans, M. Nicolaidis, S.-J. Wen, and T. Asis, "Clustering techniques and statistical fault injection for selective mitigation of SEUs in flip-flops," in *International Symposium on Quality Electronic Design (ISQED)*. IEEE, mar 2013.

[10] F. Augusto da Silva, A. C. Bagbaba, S. Hamdioui, and C. Sauer, "Combining fault analysis technologies for ISO26262 functional safety verification," in *2019 IEEE 28th Asian Test Symposium (ATS)*. IEEE, dec 2019.

[11] D. Mueller-Gritschneder, M. Greim, and U. Schlichtmann, "Safety evaluation based on virtual prototypes: fault injection with multi-level processor models," in *2016 International Symposium on Integrated Circuits (ISIC)*. IEEE, dec 2016.

[12] B.-A. Tabacaru, M. Chaari, W. Ecker, T. Kruse, and C. Novello, "Fault-effect analysis on system-level hardware modeling using virtual prototypes," in *2016 Forum on Specification and Design Languages (FDL)*. IEEE, sep 2016.

[13] M. Singh and S. Kim, "Security analysis of intelligent vehicles: Challenges and scope," in *2017 International SoC Design Conference (ISOCC)*. IEEE, nov 2017.

[14] G. Kalamkar, A. Gotkhindikar, and A. R. Suryawanshi, "Low-level memory attacks on automotive embedded systems," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, aug 2018.

[15] G. Kornaros, O. Tomoutzoglou, and M. Coppola, "Hardware-assisted security in electronic control units: Secure automotive communications by utilizing one-time-programmable network on chip and firewalls," *IEEE Micro*, vol. 38, no. 5, pp. 63–74, sep 2018.

[16] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 11: Guidelines on application of ISO 26262 to semiconductors*, International Standardization Organization Std., Dec. 2018.

[17] Renesas, "R-Car M3 Automotve SoC specification," 2020. [Online]. Available: https://www.renesas.com/us/en/solutions/automotive/soc/r-car-m3.html

[18] Infineon, "AURIX Family - TC264DA," 2020. [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/aurix-family-tc264da-adas/

[19] Texas Instruments, "TDA2SG SoC processor for ADAS applications," 2020. [Online]. Available: http://www.ti.com/product/TDA2SG

[20] ISO, *ISO 26262 Road Vehicles - Function Safety - Part 5: Product development at the hardware level*, International Standardization Organization Std., Dec. 2018.

[21] ISO, "ISO 21434 Road vehicles - Cybersecurity engineering." [Online]. Available: https://www.iso.org/standard/70918.html

[22] *Amber 2 Core Specification*, opencores.org, Mar. 2015.

[23] Cobham Gaisler, *LEON3 Multiprocessing CPU Core*, 2010.

[24] D. Lampret et al., *OpenRISC 1000 Architecture Manual*, revision 0 ed., opencores.org, Dec. 2012.

[25] OpenRISC, "OpenRISC Community," 2020. [Online]. Available: https://github.com/openrisc

[26] N. Kanekawa, T. Meguro, K. Isono, Y. Shima, N. Miyazaki, and S. Yamaguchi, "Fault detection and recovery coverage improvement by clock synchronized duplicated systems with optimal time diversity," in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*. IEEE Comput. Soc, 1998.

[27] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design & Test of Computers*, vol. 27, no. 3, pp. 4–19, may 2010.

[28] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 744–754, mar 2016.

[29] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. Sonza Reorda, "Test program generation for communication peripherals in processor-based SoC devices," *IEEE Design & Test of Computers*, vol. 26, no. 2, pp. 52–63, mar 2009.

[30] A. Floridia, E. Sanchez, and M. Sonza Reorda, "Fault grading techniques of software test libraries for safety-critical applications," *IEEE Access*, vol. 7, pp. 63 578–63 587, 2019.

[31] M. Gaudesi, I. Pomeranz, M. Sonza Reorda, and G. Squillero, "New techniques to reduce the execution time of functional test programs," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1268–1273, jul 2017.

[32] E. Sanchez, "Increasing reliability of safety critical applications through functional based solutions," in *2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE, apr 2018.

[33] P. D. Schiavone, E. Sanchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An open-source verification framework for open-source cores: A RISC-V case study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, oct 2018.

[34] A. Floridia *et al.*, "Deterministic cache-based execution of on-line self-test routines in multi-core automotive system-on-chips," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020. Accepted for publication, to appear.

[35] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. Sonza Reorda, and J.-G. Mess, "Improved test solutions for COTS-based systems in space applications," in *VLSI-SoC: Design and Engineering of Electronics Systems Based on New Computing Paradigms*. Springer International Publishing, 2019, pp. 187–206.