Automatic Generation of System-Level Test for Un-Core Logic of Large Automotive SoC

Francesco Angione, *Member, IEEE*, Paolo Bernardi, *Senior Member, IEEE*, Giusy Iaria, *Member, IEEE*, Claudia Bertani, and Vincenzo Tancorre

***Abstract*—Traditional structural tests are powerful automatic approaches for capturing faulty behavior in integrated circuits. Besides the ease of generating test patterns, structural methods are known to be able to cover a vast but incomplete spectrum of all possible faults in a System-on-Chip (SoC). A new step in the manufacturing test flow has been added to fill the leftover gaps of structural tests, called the System-Level Test (SLT), which resembles the final workload, and environment. This work illustrates how to build up an automated generation engine to synthesize SLT programs that effectively attack structural test weaknesses from both a holistic and an analytical perspective. The methodology targets the crossbar module, as one of the most critical areas in the SoC, and it simultaneously creates a ripple effect across the un-core logic. Experimental results are conducted on an automotive SoC manufactured by STMicroelectronics.**

***Index Terms*—Automotive system-on-chip (SoC) testing, manufacturing testing, system-level test (SLT), functional programs, design-for-testability (DfT).**

I. Introduction

STRUCTURAL tests, such as those based on Scan Chain and Logic/Memory Built-In Self-Test (BIST) architectures, are compelling automatic approaches for capturing faulty behaviors in Systems-on-Chip (SoCs). Nonetheless, it is known that structural tests may generate test escapes [1], [2], i.e., faults not captured by the structural pattern set may affect devices shipped to market and arise along the chips' mission. The reasons behind this substantial limitation of structural tests are various. Some generation constraints, such as the limited pattern generation time, impact the effectiveness of the Automatic Test Pattern Generator (ATPG) commercial tools. Consequently, the amount and quality of the ATPG-generated patterns are often determined as a balance between the fault coverage and the CPU time. Automatic Test Equipment (ATE) may impose extra constraints. ATEs are expensive machinery that apply the test patterns to the SoC during the manufacturing test flow. ATEs are often limited in memory for pattern storage. Hence, ATPG patterns may be further cut out from the final test recipe because of the ATE limits in pattern memory.

In case the ATE usage time per chip is contingent and hence limited to a fixed amount of milliseconds or seconds per device, another drop in coverage is highly probable [3], [4]. The combination of the aforementioned factors may lead to a large amount of structurally undetected faults leading to manufacturing test escapes. For example, in a SoC with 200 M faults (ca. 100 M gates), and 99% of fault coverage, 2 M faults are not tested.

For the category of large-sized SoC, the Design-for-Testability (DfT) architecture and configuration may introduce additional issues to reach the desired fault coverage. Whereas the circuit becomes too large to test in one shot (e.g., for power reasons), the SoC is often partitioned into multiple groups of modules. Several distinct scan chains are inserted in each group of modules, or distinct Logic BIST (LBIST) islands are drawn to cover subsets of the fault universe. Such partitioning methods enable a power-effective test application, but that may also introduce some structural untestability [5], [6]. Consequently, scan chains or BIST approaches test the component's interactions to some extent [7], but they may especially fail to check the interaction of on-chip components. For example, a functional communication path between CPU registers and a given peripheral could not be tested if the CPU and such peripheral structurally belong to different scan or BIST partitions.

To overcome the issues of structural tests, which have become evident and significant in the last decade, a new step in the manufacturing test flow has been introduced just before the Final Test for filling the leftover gaps of structural tests, the so-called *System-Level Test* (SLT). The SLT is in charge of exacerbating system-level failures through functional procedures. A common holistic strategy for verifying the correct behavior of a device is to boot an Operating system [8]. The boot of an OS is expected to prepare the environment for running tasks by configuring peripherals and runtime variables in memory; thus, it is likely to exercise the interaction among on-chip components. In case of a faulty behavior, the system may return an error condition or hang.

The major limitation of SLT is the difficulty in grading its ability to cover fault; a fault simulation process to compute the fault coverage of an SLT procedure like an OS boot may

Received 23 July 2024; revised 3 July 2025; accepted 3 July 2025. Date of publication 10 July 2025; date of current version 11 August 2025. This work was supported by the Italian Ministry of Research and University under Grant DM1061. Recommended for acceptance by D. Gizopoulos. *(Corresponding author: Francesco Angione.)*

Francesco Angione, Paolo Bernardi, and Giusy Iaria are with the Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy (e-mail: francesco.angione@polito.it; paolo.bernardi@polito.it; giusy.iaria@polito.it).

Claudia Bertani and Vincenzo Tancorre are with STMicroelectronics, 20864 Agrate Brianza, Italy (e-mail: claudia.bertani@st.com; vincenzo.tancorre@st.com).

Digital Object Identifier 10.1109/TC.2025.3587515

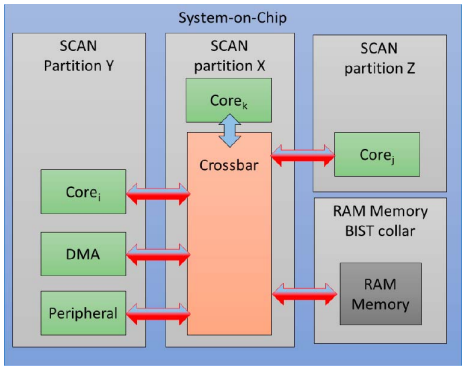


Fig. 1. Crossbar inside a scan partition.

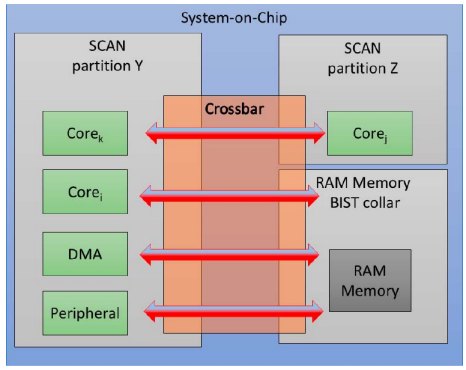


Fig. 2. Crossbar shared between scan partitions.

require tremendous CPU efforts and extensive time. In addition, given the functional nature of SLT, currently, there are almost no solutions to automate its generation, which is left to the experience of skilled test engineers who manually assemble the SLT suite (or library).

Due to structural test limitations, this work focuses on developing a System Level Test (SLT) application tailored for testing the un-core modules of SoCs. In particular, it takes the crossbar as the target module because it serves as the highway between various un-core logic in the SoC, which is critical for ensuring seamless communication.

Holistically speaking, a good test for the un-core logic modules must activate a large amount of the SoC modules; a proper SLT mimics the behavior of an OS boot that sets up the system through the activation of the chip's functional units (i.e., CPU, memories, DMA, peripheral cores, etc.) [8]. By handling the communication between different on-chip components, an SLT program running on a faulty chip could route compromised data or execute communications in a wrong manner, such as reaching the incorrect modules, corrupting the communication control signal, or even triggering unintended priority changes when the chip permits complex master/slave bus contention schemes.

From an analytical perspective, considering the crossbar as the SLT target permits to effectively deal with DfT issues. The crossbar module can reside in a scan partition or an LBIST island, as Figs. 1 and 2 show.

Figs. 1 and 2 depict a couple of possible scenarios where the SoC modules are bounded into two or three scan partitions. Because the scan partitions are operated individually and mutually, e.g., one at a time, it is likely that the glue logic gate pinched between LBIST islands may be either not exercised due to not driven signals from an inactive island, or their behavior is not captured because it could only be propagated to an inactive island. Similarly, MBISTs may introduce structurally untestable logic between the so-called MBIST collar and the actual RAM array [9]. Fig. 1 illustrates the case when the crossbar is fully included in a single DfT partition, while Fig. 2 depicts the case when the crossbar gates are distributed over several DfT partitions.

In both cases, DfT partitions may provoke some fault to become very hard to detect. Blue/red arrows visualize the intrinsic weaknesses introduced by the DfT partitions. An effective SLT suite for the crossbar is therefore a good candidate to cover such shadowed zones of the circuit.

The proposed approach not only facilitates effective testing of the crossbar itself but also extends to the connected logic [10], including interfaces to/from cores, and un-core logic, such as peripheral bridges, peripherals, and memory controllers, etc.

Overall, this paper brings some innovative contributions to highlighting the necessity of SLT and to automate the creation of SLT programs:

- An automatic generation approach is proposed to create effective SLT libraries for un-core logic based on the architectural information of the chip; by focusing on generating an SLT application designed to test the crossbar module effectively, it simultaneously creates a ripple effect across the un-core logic of the SoC.
- A methodology is illustrated to classify faults uncovered by structural tests (residual structural faults) and quantify the impact of DfT configuration and usage on the structural coverage; such a contribution introduces guidance in the SLT generation and minimizes grading efforts.

The proposed generation methodology is netlist independent and relies on the high-level description of the system, which can be obtained from the SoC documentation. This generation methodology is flexible and automated because supported by an abstracted graph-based model of the SoC, where SoC modules are represented as nodes and their means of communication as edges between nodes (i.e., the CPUs, DMAs, and peripherals). An algorithm visits the graph and synthesizes functional program chunks by generating a list of addresses where read/write operations are performed for each bus master.

For the sake of properly grading an SLT suite written by following a holistic philosophy, the paper also describes how to perform a thorough algorithmic analysis of the residual fault list after structural patterns coverage, i.e., faults not detected by structural tests. Such information turns out to be crucial to validate the effectiveness of the SLT programs.

The effectiveness of the proposed generation methodology to cover SLT-oriented faults is supported by experimental results obtained on an automotive SoC manufactured by STMicroelectronics. The SLT suite is compared with an RTOS boot in terms of fault coverages.

The two crossbars weigh respectively about 78k and 56k gates. The number of faults for the crossbars are ca. 155k and 111k, respectively. Structural tests include about 150k scan-based test patterns that reach a Stuck-At Fault (SAF) coverages of 95.18 % and 94.28%, meanwhile, the Transition Delay Faults (TDF) coverages are 88.71% and 87.49%, respectively for the crossbars.

The SAF coverage is increased by 1.9% and 1.67%, bringing the crossbars coverages up to 97.08% and 95.95% respectively. Finally, TDF coverage is raised of 2.4% and 2.27% respectively, and compared to structural tests, the coverages goes up to 91.11% and 89.76%.

The generated SLT suite has been assessed on different SoC modules such as CPU (1.5M SAF/TDF), peripheral bridge (76k SAF/TDF), Controller Area Network (CAN) peripheral (435k SAF/TDF), and the overall memory controllers and collars (1.4M SAF/TDF).

A deep dive into the SoCs netlist returned a classification of the untested faults at the structural level. This information is crossed with fault simulation results of the SLT suite for stuck-at and transition-delay fault models.

The paper is structured as follows: in Section II, basic concepts on manufacturing test flow are illustrated, in Section III is presented the graph model, and it depicts the used methodology. Section IV highlights experimental evidence on a large automotive SoC. Section V draws some conclusions.

II. Background

A. Manufacturing Test Flow

As the transistors scale, the density of transistors per area increases [11]. This important achievement led to an exponential rise up in the complexity of integrated circuits, and the testing scenario becomes more complex [11].

As previously mentioned, structural tests (scan-based) have been introduced to reduce the testing complexity flow and increase the automation of generating test patterns for a given integrated circuit [12]. From the introduction of scan-based structural approaches, the generic manufacturing test flow has been stabilized, with minor changes in the order of test steps, which is company-dependent.

The test flow is divided into different stages, each one of them targeting different defects and with a specific goal to discard manufactured faulty devices [8], [13]:

- Wafer Test done at the wafer level checks for the primary electrical functionalities of the chip and executes structural test patterns.
- Package Test done at the package level measures and tests the essential electrical characteristics of the pins.
- Burn-In, mainly for automotive and safety-critical devices, for exacerbating latent defects [14], [15]. In this way, after the Burn-In phase, devices manufactured with weaknesses can be captured by later test steps.
- System-Level Test, it has been added as an additional test for safety-critical and automotive devices [1], to verify the correctness of devices by resorting to complex functional programs.
- Final Test can detect faults by applying a mix of structural and functional tests.

In the last decade, some companies started to introduce an additional test step in the manufacturing test flow [2], [16], [17], the so called *System-Level Test* (SLT), it mainly consists in running functional applications that mimic the in-field behavior of the device. For example, a holistic baseline benchmark is to boot and run a Real-Time Operating System (RTOS) with some workloads. A grey box SLT program generation method has been proposed to automatically create code snippets that control the Device Under Test (DUT)'s power consumption, outperforming handcrafted snippets in experiments on a RISC-V processor by leveraging mutation-based fuzzing [18], [19]. Another approach utilizes LLMs with Structural Chain of Thought (SCoT) prompting and a pool of prior snippets to generate high-quality SLT code targeting non-functional properties like Instruction Per Clock (IPC), improving snippet quality and reducing manual effort [20].

The proposed generation SLT methodology described in this work paper stands out for its automated, and high-level approach to test generation. While related frameworks like those in [18], [19] and [20] share some similarities in automation and high-level modeling, they excel in optimizing SLT programs for non-functional properties like IPC or power consumption; the SLT methodology's focus on un-core logic, fault classification, and system-level interactions makes it uniquely suited for modern SoC testing challenges.

In addition, it is important to highlight that within the un-core logic, there are also communication peripherals that must be tested effectively. In the literature, various methodologies already exist, including a manually deterministic methodology [21], a hybrid methodology [22] based on automated and deterministic approaches, and online testing methodologies [23] that can be re-used at manufacturing. However, the aforementioned methodologies lack system-level interaction between the on-chip and off-chip components (CPU, memories, the external world, and other peripherals), and they do not address the structural test weaknesses discussed by the authors in [24] during manufacturing testing. The proposed automatic generation flow can be configured to implement the deterministic methodology described in [24] for effectively testing communication peripherals, combined with the required companion module.

B. Trade-Offs of Scan-Based Testing in Large SoC

Scan-based testing is essential for achieving high fault coverage in large SoCs but nevertheless presents challenges in the context of ever-growing design complexity. Automatic Test Pattern Generation (ATPG) strategies for scan-based testing in complex designs can be computationally expensive, requiring multi-model incremental generations [25] to reach high fault coverage needed by devices intended for safety-critical sectors.

Although the ATPG seeks to achieve complete fault coverage, many patterns can result in overly broad test patterns, which can configure the circuit in ways not actually used by the device during its operational mode [26]. This over-testing can increase test time and computational resources [27]. Moreover,

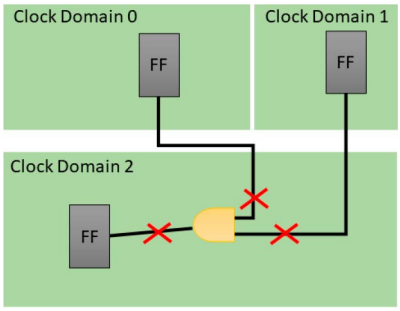


Fig. 3. Cross domain faults example.

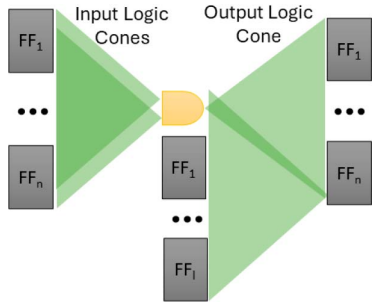


Fig. 4. ATPG fault excitement and propagation.

in large SoC, ATPG could leave some faults uncovered [1]. Thus, the scan-based pattern set is not always exhaustive.

A possible weakness of scan-based patterns could lay in logic gates that are controlled by scan flops belonging to one clock domain island (or scan chain configuration) and observed by scan flops belonging to another clock domain island (or scan chain configuration). Such a scenario is depicted in Fig. 3, where the shown gate in yellow is controlled by scan flops belonging to two clock domains that are both different from the clock domain where the observer scan flops is. Faults related to gates with the mentioned characteristics could be challenging to test for ATPG in a single run, and they require multiple generation using different combination of configurations or dedicated strategies [5], [6].

In literature, the algorithms behind the ATPG computation are varied [28], and typically heuristic-based; one of the most common is the FAN Algorithm [29], which focuses the generation on the gates' fan-out. Its aim is exciting and propagating faults with large input/output logic cones, i.e., it tries to maximize the probability of capturing faults in observable FFs thanks to cascading effects.

In addition, commercial ATPG tools also have trace-back functionality, which is therefore also critical for circuit debugging and analysis. The proposed methodology, which will be detailed in Section III, makes use of these abilities of the ATPG tools to perform a structural analysis of the remaining fault locations not covered by the pattern set scan in use.

To better detail what an ATPG tool could face, Fig. 4 represents a possible scenario. The fault on the gate (in yellow) output must be excited, i.e., the ATPG must create the values to be loaded in the FFs from 1 to n. As a second important step, the fault must be observed in the FF of the output logic cone; simultaneously, the ATPG must create the constrained values to be loaded in different FFs to propagate the excited fault to observable FFs. Generally, the output logic cone ends in a high number of FFs, and as a consequence, a fault can be observed as a cascade effect of a different test pattern. On the other hand, if the output logic cone ends into a single observable FF, the ATPG must invest efforts to generate a specific test pattern to excite and propagate a given fault, consequently increasing the computing time and the test pattern set size for a single, specific, fault.

In the last decades, complementary strategies such as functional testing and system-level tests have been used in conjunction with scan-based testing in order to address scan-based weaknesses [1], [2], [8], [16] without any netlist-based structural analysis but instead on empirical observation of failures in faulty devices running in-field like application during production.

III. Proposed Methodology

A malfunctioning crossbar logic could send data down the wrong on-chip components, delivering it to unintended recipients (masters) and potentially changing the order of importance (priority) for different masters. SLT applications targeting the crossbar logic would simulate real-world interactions between on-chip components to verify unintended escaped faults from structural tests. By generating SLT applications for the crossbar module, it creates a ripple effect on the un-core logic.

In addition, a major challenge in developing these tests, or any general functional test program, is automation. The proposed methodology aims to fill the gap in terms of automatic test generation by providing an automatic generation methodology for SLT applications relying on an abstracted model of the SoC.

Fig. 5 shows the generation methodology. The proposed method is based on a *Grind Nexum algorithm* that exploits an internal graph representation of all the components in the SoC, including CPUs, DMAs, and peripherals. The graph is built by starting from a memory map description. Meanwhile, crossbars master and slave ports are modeled as a virtual vertex to allow a more leisurely visit of the model. In the approach, the devised graph data structure is visited to cover all the edges or nodes (i.e., the connections between each component), and to produce a list of addresses and functions for which masters read and/or write to slaves memory locations.

The targeted addresses are generated as a data structure encapsulated into a library called *System Software Test Library* (S-STL or S^2TL). The aforementioned library can be integrated into a Real-Time Operating System or a bare metal application and executed on the Design Under Test (DUT).

The following subsections describe the proposed SoC model, the *Grind Nexum algorithm*, and the S^2TL.

A. Proposed System-on-Chip Model

The presented work is based on abstracting the architectural components of the SoC to vertexes of an undirected graph. Every component of the SoC is represented with a vertex in

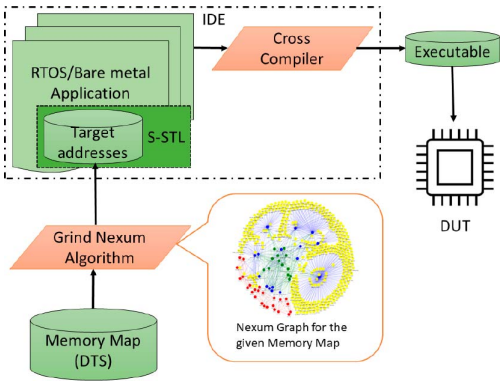


Fig. 5. Proposed methodology workflow.

the graph connected to a virtual vertex representing the master/slave port of the crossbar or a bus, where every master is connected to every slave. Eventual internal memory elements in the masters are further added to the model, representing a fully connected internal graph.

More in detail, a CPU may contain a private cache and RAM strongly tied to the CPU registers. Therefore, its graph-based representation has to take into account all those connections.

The proposed SoC model can be analytically represented with the following formulas, where X denotes the bipartite graph between virtual masters and slaves, and N the graph produced by the union of the graph X, and the graph representing the peripherals, crossbar slaves and masters ports, memories, cores and the rest of the un-core logic (considered as master and slave vertex).

$$X = (P_{Master}, P_{Slave}, C) \tag{1}$$

$$C = \{\forall x \in P_{Master} \exists (x, y), x \neq y, \forall y \in P_{Slave}\} \tag{2}$$

Equation 1 represents the bipartite graph modelling the crossbar master ports (P_{Master}) and the slave ports (P_{Slaves}). Meanwhile, Equation 2 models all the edges from every master to every slave without any self-loop.

$$N = \cup\{(V, E), X\} \tag{3}$$

$$V = \left\{ \bigcup_{i=0}^{M} Master_i, \bigcup_{j=0}^{S} Slave_j, \bigcup_{k=0}^{L} ResRegion_k \right\} \tag{4}$$

Equation 3 describes the union of the graphs modeling the crossbar and the other components (memories, cores, peripherals, etc.). In equation 4, there is the set of all possible vertexes of the SoC, the CPU-addressable reserved region and slaves, and the vertexes capable of acting as a master on the crossbar.

$$Master_i = \{\forall x, y \exists (x, y), \exists (x, y, z) | z \in X\} \tag{5}$$

As Equation 5 shows, a crossbar master can be composed of several sub-vertexes (*x,y*) such as internal registers, internal memories and internal caches with self-loop capabilities. In addition, internal vertexes of a given master are fully connected, and a vertex exists within a master, which is also present in the graph described by Equation 1. Regarding the *Slaves*, or *Reserved Region*, Equation 6 shows how the other sets are built.

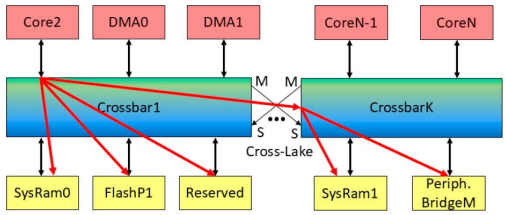


Fig. 6. Example architecture with N masters (red), M slaves (yellow), and K crossbars interconnected by a cross-lake.

$$Slave_j = \{\forall x, y \exists (x, y), \exists (z, x, y) | z \in X\} \tag{6}$$

$$E = \{\exists (x, z, k, y) | x, y \in V, z, k \in X, \tau(x, y) \neq \emptyset\} \tag{7}$$

On the other hand, Equation 7 models the edges between a master and a slave, including the virtual vertexes of the master and slave ports of the crossbar and their type of operation (i.e., how data can be transferred) described by Equation 8 depending on the vertex type.

$$\tau(x,y) = \begin{cases} \text{Cpu} & \text{x a CPU regs, y CPU regs} \\ \text{Peripheral} & \text{x a CPU regs, y a Peripheral} \\ \text{Flash} & \text{x a CPU regs, y a Flash} \\ \text{Ram} & \text{x a CPU regs, y a Ram} \\ \text{Ram\_fetch\_instr} & \text{x a ICache, y a Ram} \\ \text{Flash\_fetch\_instr} & \text{x a ICache, y a Flash} \\ \text{Icache} & \text{x a ICache, y generic} \\ \text{Dcache\_exception} & \text{x a DCache, y a Reserved/Peripheral} \\ \text{Dcache} & \text{x a DCache, y generic} \\ \text{DMA\_32\_transfer} & \text{x a Ram, y a Peripheral} \\ \text{DMA\_No\_wait} & \text{x a Ram, y Reserved} \\ \text{DMA} & \text{x a Flash, y a Ram} \end{cases} \tag{8}$$

The conditions for the type of operation retrieved from Equation 8 are specular to provide the correct operation type, i.e., $\tau(x, y) = \tau(y, x)$. The order implies only a read or write operation. For example, a *Peripheral* type of operation when x is a CPU register and y is a peripheral, it is a write to the peripheral control registers. Meanwhile, when x is a peripheral and y is a CPU register, it is a read to the peripheral control registers. Additional care is needed if a CPU instruction master port is present. In order to be exercised, the executed code must be relocated to the given address.

For example, suppose to have an SoC Architecture as the one represented Fig. 6; every Core has an internal private cache and, naturally, CPU registers, and it is connected to the crossbar through a single master port.

The presence of virtual vertexes representing the master/slave port of the crossbar enables further analysis as well as the creation of all the possible connected paths from a master to all addressable slaves, as it happens architecturally. For example, in

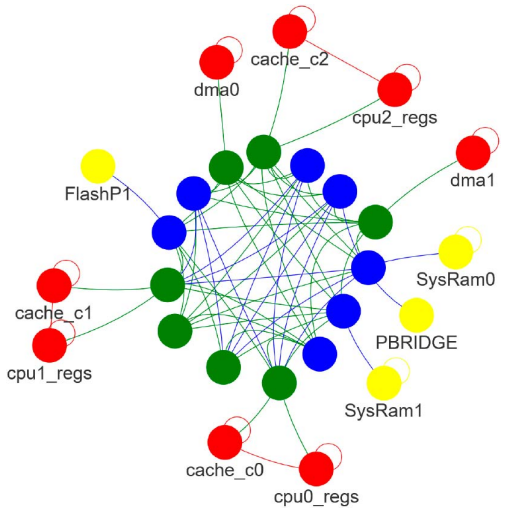


Fig. 7. Generated graph model for the example SoC architecture with two linked crossbars, crossbar master ports (green) and slave ports (blue).

Fig. 6 red arrows represents all the possible logical paths on the crossbar from master *Core2*. Therefore, the graph model generated from Fig. 6 is shown in Fig. 7 using only two crossbars connected by a crosslake for readability purposes.

For the sake of this work, the generated graph is referred to as *Nexum graph* (in Latin, *Nexum* means connections), representing the connection between every component in the SoC.

Vertexes in Fig. 7 are colored depending on the nature of components:

- Red, vertex (component) capable of acting as master on the crossbar.
- Green, virtual vertex modeling the master bus port.
- Blue, virtual vertex modeling the slave bus port.
- Yellow, vertex (component) capable of acting as a slave on the crossbar.

The above reasoning can be easily extended to any number of masters and slaves, independently from the technology or the bus protocol. If a component is simultaneously master or slave, master color has the predominance. The proposed SoC model is used in the proposed methodology to analyze and generate a SLT application for testing the un-core logic for each possible master read/write operations on memory addresses (RAM, ROM, peripherals, and reserved partitions) through the crossbar.

B. Grind Nexum Algorithm

The *Grind Nexum algorithm* is responsible for creating a graph-based representation of the SoC starting from a memory map file. Afterward, it visits the graph to cover all the nodes or edges and generates a list of target addresses and functions to execute on the DUT. In the following subsections, every algorithm step is described in detail.

1) Creating the Model: The Memory map file can be generated from the user manual with manual efforts, or exploiting generative AI, to model memory regions (for slaves) and master present in the SoC; the output format can be a custom CSV file. The manual generation of the SoC memory map file is time-consuming and error-prone. To address this challenge, *Device Tree Source* (DTS) files can be used, which are a standardized method for describing SoC hardware in the embedded Linux domain [30]. DTS files are typically already produced and validated as part of the platform development process, making them a reliable foundation for representing SoC information. This approach not only reduces manual effort but also minimizes the risk of errors during the creation process. By utilizing DTS, the representation and usage of SoC information become more streamlined and scalable, offering a robust solution for modeling hardware efficiently.

The Memory map file acts as a comprehensive hardware description document, detailing various components such as CPUs, memory regions, peripherals, and other devices. It includes specific information about address ranges, sizes, access types, clock frequencies, interrupt mappings, and other device-specific settings, such as the attached master or slave ports and other possible communication paths between on-chip components. This level of detail is essential for the proper initialization and operation of the hardware. One of the key functions of a Memory map file is to describe non-discoverable hardware, including custom peripherals and specific memory mappings. The file is written in a human-readable format, which allows developers to easily customize and modify the hardware description as needed, facilitating debugging and maintenance of the hardware configuration. This approach simplifies the process of hardware configuration and ensures that the system can correctly interact with all its components. Overall, this Memory map file serves as a crucial blueprint for the modeling an SoC, providing a clear and detailed representation of the system's hardware layout, capabilities, and interconnections.

All the information contained in the Memory map file is consolidated by Algorithm 1 to construct the SoC's undirected graph model, known as the Nexum Graph. The algorithm creates a vertex for each addressable region—both reserved and non-reserved—embedding key attributes such as address ranges, sizes, etc. (lines 3–17 of Algorithm 1). Afterward, it establishes functional links between on-chip components using defined communication paths (lines 22-32). Since these functional communication paths are bidirectional (with crossbar masters able to both read and write), the resulting SoC graph model is undirected.

It is important to emphasize that the master, slave, and crossbar ports of the crossbars are modeled as virtual nodes in the graph, effectively representing an addressable region within the SoC's entire address space. The graph generation process begins with the construction of an internal crossbar graph where every master node is connected to every slave node. Once this initial connectivity is established, the algorithm proceeds to link each master and slave node to their corresponding crossbar ports. This two-step process, starting from the comprehensive master-to-slave connectivity and then integrating the crossbar ports (as shown in lines 15–17 of Algorithm 1), ensures that the final Nexum graph accurately encapsulates both the functional and physical interconnections of the SoC.

Algorithm 1 Nexum Graph model generation

Require: Compliant Memory Map File
1: xbar_graph $= \emptyset$, nexum_graph $= \emptyset$
2: Load memory map from file in mem_map
3: **for each:** $mem\_region \in mem\_map$
4: **if** Type(mem_region)=XBAR **then**
5: $add\_node(mem\_region, xbar\_graph)$
6: **else**
7: $add\_node(mem\_region, nexum\_graph)$
8: **end if**
9: **if** Type(mem_region)=CPU **then**
10: $cores\_edges[mem\_region].append(mem\_region)$
11: **end if**
12: **if** $Type(mem\_region) = no\_xbar\_connection$ **then**
13: $no\_xbar\_connection.append(mem\_region)$
14: **end if**
15: **for each:** $P_{master} \in xbar\_graph$
16: **for each:** $P_{slave} \in xbar\_graph$
17: $xbar\_graph.add\_edge(P_{master}, P_{slave})$
18: **for each:** v $\in$ nexum_graph
19: **if** $Type(v) = RAM$ **then**
20: $nexum\_graph.add\_edge(v, v)$
21: **end if**
22: **for each:** link $\in$ v.masters_slaves_ports
23: $links = v.masters\_slaves\_ports[link]$
24: $nexum\_graph.add\_edge(v, links, xbar\_graph)$
25: **for each:** core $\in$ $cores\_connection$
26: **for each:** $v_1 \in cores\_connection[core]$
27: **for each:** $v_2 \in cores\_connection[core]$
28: $nexum\_graph.add\_edge(v_1, v_2)$
29: **for each:** $no\_xbar\_connection \in no\_xbar\_connections$
30: $v_1, v_2 = no\_xbar\_connection$
31: $nexum\_graph.add\_edge(v_1, v_2)$
32: **return** $nexum\_graph$

Algorithm 2 Node Visit Function for a Node set (masters or slaves)

Require: $Nexum\_Graph, Xbar\_Graph$
Require: $Node\_Set, Target\_Set, Action\_Direction$
1: **for each:** $node \in Node\_Set$
2: root = node
3: **if** $root \neq Reserved$ & $root \notin Xbar\_Graph$ **then**
4: $actions[root] = []$
5: $visited = set(); sink = None$
6: $visited.add(root)$
7: $connections = Nexum\_Graph[root]$ ▷ Nodes connected to root
8: **while** $connections \neq \emptyset$ **do**
9: $current\_node = connections.pop()$
10: **if** $current\_node \in visited$ **then**
11: **continue**
12: **end if**
13: $visited.add(current\_node)$
14: **if** $current\_node \in Target\_Set$ **then**
15: $sink = current\_node$
16: **if** $Action\_Direction ==$ "$master\_to\_slave$" **then**
17: $actions[root].append(Action(root, sink))$
18: **else**
19: $actions[sink].append(Action(sink, root))$
20: **end if**
21: **end if**
22: $connections.extend(Nexum\_Graph[current\_node])$
23: **end while**
24: **end if**
25: **return** $actions$

Moreover, there are special nodes that require additional care. For example, CPUs may contain additional subunits such as cache, local private memories, and CPU registers. These components are represented as vertices in the graph and are associated with the same CPU master port of the crossbar. They are interconnected by dedicated functional communication paths (bypassing the crossbar) that enable direct data transfer, as illustrated in line 25 of Algorithm 1. Furthermore, these nodes feature self-loops, demonstrating their ability to internally transfer or store data (for instance, moving data from one register to another or between memory words in RAM), as shown in lines 19–21 of Algorithm 1.

Regarding reserved memory regions, they are modeled and visited as vertex, sharing similarities within the memory region where they lay. For example, a reserved region after an addressable RAM region is considered an additional RAM vertex. In addition, in real architecture, there are reserved master and slave ports in the crossbar. They are not modeled without any loss of generality since they will never be exercised in the field. Moreover, certain communication paths between masters and slaves bypass the crossbar and are modeled as additional edges directly in the memory map file (see lines 12–14). These extra edges are then incorporated into the final SoC graph model during its construction (see lines 29–32). This approach ensures that all direct communication routes (those not mediated by the crossbar) are accurately represented, providing a complete depiction of the data flow within the system.

2) Visiting the Model: Once the *Nexum graph* has been created, the *Grind Nexum algorithm* can visit the graph model in different ways:

- Visiting all the nodes starting from master nodes as node set and slave nodes as target set using a Deep First Search, independently from the edges (means of transfer), as Algorithm 2 describes.
- Visiting all the edges starting from master nodes using a Deep First Search, as Algorithm 3 describes.
- Visiting all the nodes starting from slave nodes as node set and master nodes as target set using a Deep First Search, independently from the edges (means of transfer), as Algorithm 2 describes.
- Visiting all the edges starting from slave nodes using a Deep First Search, as Algorithm 3 describes.

The performance complexity of visiting all the nodes, or the edges, is $O(|V| + |E|)$, with V and E being, respectively, the number of vertexes and edges in the *Nexum Graph*.

For node visits in Algorithm 2, the traversal is not carried out to explore every possible vertex in the graph in an exhaustive manner, but rather to efficiently locate target nodes (either masters or slaves). In this context, the DFS-like approach helps in quickly reaching a target node along one branch before exploring others.

In Algorithm 2, the only substantial difference in starting from master or slave nodes as node set resides in maximizing the bus traffic and contention on the crossbar ports, from masters it results that multiple slaves are exercised from the same master port, from slaves it results that multiple master are exercising at the same time the slave port.

Starting the visit from master nodes is going to maximize the traffic on the master port of the crossbar; meanwhile, starting the visit from slave nodes is going to maximize the traffic on the slave port connected to a given slave, i.e., multiple masters are accessing at the same time the same slave, and as a consequence, they must be properly synchronized.

Algorithm 3 Combined Edge Visit Function for Masters and Slaves

```
Require: Nexum_Graph, Xbar_Graph
1: for each: node ∈ (masters ∪ slaves)
2: root ← node
3: if root ≠ Reserved ∧ root ∉ Xbar_Graph then
4:     actions[root] ← []
5:     visited_nodes ← ∅
6:     visited_edges ← ∅
7:     visited ← {root}
8:     sink ← None
9:     stack ← [root]
10:    edges ← ∅
11:    while stack ≠ ∅ do
12:        current_node ← stack.pop()
13:        if current_node ∉ visited_nodes then
14:            edges[current_node] ← Nexum_Graph[current_node].edges()   ▷ Retrieve the edges of the current node
15:        end if
16:        if edges[current_node] = ∅ then
17:            stack.pop()   ▷ No more edges from the current node, so backtrack
18:        else
19:            edge ← next(edges[current_node])
20:            if edge ∉ visited_edges then
21:                visited_edges.add(edge)
22:            end if
23:            if (root ∈ masters ∧ current_node ∈ slaves) then
24:                sink ← current_node
25:                actions[root].append(Action(root, sink))
26:            else if (root ∈ slaves ∧ current_node ∈ masters) then
27:                sink ← current_node
28:                actions[sink].append(Action(sink, root))
29:            end if
30:            sink ← current_node
31:            stack.push(edge[0], edge[1])
32:        end if
33:    end while
34: end if
35: return actions
```

At the end of a visit, the algorithms generate a list of *Actions*. An *Action* is a defined data structure containing a source address, a destination address, and a function pointer to be applied to the source and destination address on the DUT. The resolution of the function pointer is made accordingly to Equation 8 (i.e., it depends on the source and destination node's type). The *Nexum Graph* is an undirected graph. Therefore, for each *Action*, read and write operations are generated between source and destination. Furthermore, every node contains a memory range of a given size that would be impossible to read and write entirely for every node in the *Nexum Graph*. In order to effectively address this issue, based on the approach proposed in [31], addresses are generated at half of a given address range for each source and destination, and in that interval of words read and write operations are carried out. Regarding peripherals, read and write operations are generated according to their addressable registers.

A detailed view of the algorithm for visiting all the nodes from multiple masters can be seen in Algorithm 2.

First, all the functional paths from the *Root* master to every slave are visited. Afterward, the tool can start the visit from another master with the same approach, visiting all the slaves and generating *Actions* accordingly.

An important aspect to consider is the maximization of concurrency that could happen on the bus. In order to maximize the concurrency, internal paths from the virtual master port to the virtual slave port of the crossbar are exercised multiple times and concurrently by different masters.

Meanwhile, node maximization could ensure that all the functional units are touched at least once. It does not guarantee the functional paths to and from the bus, especially on those nodes where a self-loop exists, e.g., RAMs. A DMA or a CPU could exploit RAM self-loops to maximize the bus traffic. Self-loops are not considered in the node visit. Instead, they can be visited from the edge visit represented in Algorithm 3.

More in detail, edge visit algorithms are exploring the *Nexum Graph* until they find a slave or no more edges to push on the stack data structure. In particular, in Algorithm 3, in lines 25, the edge data structure comprises a tuple of two nodes to be added to the stack structure.

In Algorithm 3, the idea is to generate the minimum number of trees to touch at least once every edge. In this strategy, the algorithm processes edges sequentially within each node's list, while the DFS stack governs the overall traversal order. From a testing perspective, this means that functional paths, such as a CPU-to-RAM connection and a CPU-to-Peripheral connection are treated equivalently, even though the DFS approach imposes a specific order. Importantly, the ordering of edge visits can be adjusted as needed to prioritize or specifically isolate certain paths for testing purposes. In this way, the methodology can ensure that all the DUT functional paths and on-chip components are exercised.

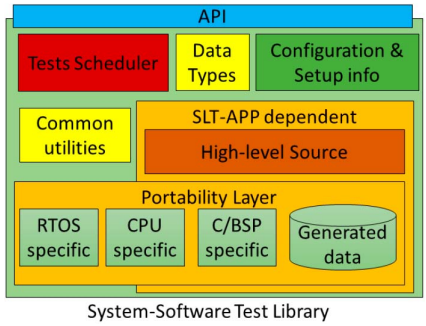


Fig. 8. Software modules decomposition of a S^2TL.

C. The System-Software Test Library

This section aims to provide a general software architecture for a *System Software Test Library* (S-STL or S^2TL) in order to reduce as much as possible the porting time or development time and, at the same time, let test engineers focus only on the functional test program itself. Fig. 8 shows the organization of the library mentioned above.

Fig. 8 depicts a clear separation between high-level functions and data, data types, and configuration directives, which are all SoC and ISA independent from the SoC and ISA dependent part in the portability layer. The portability layer contains CPU-specific functions, a board/chip support package (C/BSP), and Real-Time Operating System services. Following this separation, modifying only the portability layer when migrating to a different SoC architecture or ISA an SLT application is sufficient.

Algorithm 4 Pseudo code for a generic Action

```
Require: Source, Destination, Error Pointer
Ensure: Test data in Source
Move Test Data from Source to Destination
Read Test Data from Destination
if Data are not saved correctly or (generated Exception and Expecting Exception) then
    return notOK
end if
return OK
```

By organizing the library as Fig. 8 shows, it is possible to easily integrate further SLT approaches by providing the necessary software without rewriting the functional test program from scratch. The library must encapsulate a runtime test dispatcher for all the available cores. Despite the separation of internal modules, the S^2TL can be accessed only by a bare metal application or an RTOS through the Application programming interface (API).

Within the scope of this paper, the S^2TL implements the runtime test dispatcher specific for the generated target address by the aforementioned methodology. It includes parsing the generated data structure in a high-level programming language and calling the proper function pointers. It also synchronizes the cores while executing the tests and prepares the DUT.

The so-called *Actions* are SoC and CPU dependent, and they are implemented between different sources and destinations (such as CPU registers, RAMs, DMAs, Peripherals, reserved region) in the portability layer.

Pseudo Code 4 depicts the general philosophy of an *Action*, i.e., move data from a source to a destination and verify its behavior with the expected one. The actions executed by the S^2TL primarily involve data movement between a source and a destination. However, the nature of this data movement depends heavily on the type of destination, which is why a portability layer is implemented to bind the library to the underlying architecture. For instance, data movement between CPU registers and RAM is relatively straightforward and typically does not pose significant challenges. In contrast, data movement between CPU registers and peripheral registers requires careful consideration. If the peripheral is not properly configured, such operations could trigger exceptions or unintended behavior. Similarly, when using a DMA controller for data movement, it must be correctly configured to avoid issues such as data corruption or system instability. Regarding reserved memory area, the library must capture the generated and expected exception.

Additionally, the choice of data patterns used during data movement can significantly impact fault coverage. Certain patterns may better expose faults in the communication paths or hardware components, while others might fail to trigger specific edge cases (out of scope for this work).

D. Assessment of SLT Procedures

In order to assess analytically whether the holistic-driven SLT generation is actually effective in bridging the inherent issues of structural testing, an analysis of the residual (non-detected) faults not covered by these tests is illustrated. Such an analysis is based on the concepts described in the Subsection II-B. Indeed, scan-based patterns suffer of some weaknesses in covering specific types of faults. For example, if a fault lays in the intersection of flop logic cones that are triggered by different clock domains, covering that fault would require more scan patterns, specifically generated to target those regions. The same behavior happens for faults laying in the intersection of flop logic cones belonging to different chain configurations. Details of the trade-offs of scan-based testing are described in Subsection II-B.

Algorithm 5 Finding faults between different scan chain configurations or clock domains

```
Require: gates, list of gates including inputs, outputs and corresponding endpoints
faults ← ∅
for each g in gates do
    scan_conf_in ← ∅
    for each inp in g.inputs do
        for each endp in inp.endpoints do
            chain_conf_in.add(endp.scan_conf)
        end for
    end for
    scan_conf_out ← ∅
    for each out in g.outputs do
        not_found ← FALSE
        for each endp in out.endpoints do
            scan_conf_out ← endp.scan_conf
            if endp.scan_conf not in scan_conf_in then
                not_found ← TRUE
            end if
        end for
        if not_found is TRUE then
            faults.add(out)
        end if
    end for
    for each inp in g.inputs do
        not_found ← FALSE
        for each scan_conf in scan_conf_in do
            if scan_conf not in scan_conf_out then
                not_found ← TRUE
            end if
        end for
        if not_found is TRUE then
            faults.add(inp)
        end if
    end for
end for
return faults
```

Upon such considerations, the paper proposes a categorization of those uncovered faults. Specifically, an uncovered fault could fall into one of the following categories:

A. Clock-domain crossing: faults in the intersection of flop logic cones that are not triggered by the same clock domain.
B. Chain configurations crossing: faults in the intersection of flop logic cones that do not belong to the same chain configuration.
C. ATPG critical: faults that are difficult to observe because they are linked to a few scanned flops.
D. Functionally untestable: faults that are impossible to excite, and observe during a normal functional program execution such as debug circuitry and test enables for FFs, they are identified with the approach proposed in [32] and introduced as constant constraints in the fault simulator.

The algorithms used to assign the categories are explained below. Algorithm 5 can be used for fault extraction in both clock-domain and chain configurations crossing categories.

Algorithm 6 Extracting ATPG critical faults depending on the fan-in/fan-out ratio

Require: $gates$, list of gates including inputs, outputs and corresponding endpoints
Require: $residual\_faults$, set of faults not detected by structural-based tests
$atpg\_critical\_faults \leftarrow \emptyset$
for each g in $gates$ **do**
  $fan\_in \leftarrow 0$
  for each inp in $g.inputs$ **do**
    $fan\_in \leftarrow fan\_in+$ len$(inp.endpoints)$
  end for
  $fan\_out \leftarrow 0$
  for each out in $g.outputs$ **do**
    $fan\_out \leftarrow fan\_out+$ len$(out.endpoints)$
  end for
  $g.ratio \leftarrow \frac{fan\_in}{fan\_out}$
  $faults$.add_with_ratio($g.inputs$, $g.ratio$)
  $faults$.add_with_ratio($g.outputs$, $g.ratio$)
end for
$unique\_ratios \leftarrow$ get_unique_ratios($faults$)
$tot\_occurrences \leftarrow \emptyset$
for each not_det in $residual\_faults$ **do**
  $not\_det\_ratio \leftarrow faults[not\_det].ratio$
  $tot\_occurrences[not\_det\_ratio]$.increment()
end for
$avg\_occurrences \leftarrow \frac{tot\_occurrences}{\text{len}(tot\_occurrences.\text{keys}())}$
for each $fault$ in $residual\_faults$ **do**
  if $fault.ratio \geq avg\_occurrences$ **then**
    $atpg\_critical\_faults$.add($fault$)
  end if
end for
return $atpg\_critical\_faults$

Upstream of these algorithms, it is a requirement to possess the list of circuit gates, with each of their flop endpoints, along with the information on the clock domains and chain configuration to which they belong. As for the ATPG critical category, Algorithm 6 describes the extraction based on the set of faults not covered by the structural patterns in use. This way, the algorithm extracts the faults with a higher weight than the average one.

Once all the remaining faults not covered by the structural tests are categorized, it is possible to highlight in which categories the SLT was most successful in bridging the structural weaknesses of the DfT in complex case studies. In this way, the analysis makes it possible to verify that the holistic and automatic generation of SLT programs in the proposed methodology succeeds in its intent to cover more of the critical structural testing faults.

IV. Experimental Results

A. The Case Study: An Automotive System-on-Chip

The case study is a 40 nm Automotive SoC manufactured by STMicroelectronics and compliant with the standard ISO26262 ASIL-D, with about 20 million logic gates and about 700k flip-flops. The SoC architecture from the user manual is presented in Fig. 9(a). The SoC has a multicore architecture with three 32-bit cores using the PowerPC Variable-Length Encoding (VLE) instruction set. It has 6Mbyte of Flash memory and 128Kbyte of general-purpose SRAM. There are several peripheral bridges to access a set of on/off-chip peripherals. Regarding the communication between components, they are interconnected to two, not equivalent, fast crossbar switches AHB-AMBA 2. v6 at 64-bit and capable of working up to 200Mhz. Moreover, the two crossbars are linked with a cross-lake, which allows to tie-up crossbar one master with crossbars two slaves and vice-versa.

Fig. 9 contrasts the SoC architecture from the User Manual and the generated *Nexum graph* model. Fig. 9(b) shows a medium-high number of crossbar masters combined with a very high number of crossbar slaves. All the possible communication paths (edges in Fig. 9(b)) are 827. The manual creation of the memory map file in CSV format required a total of 10 hours by a single person, including the time spent on verification and debugging. In contrast, by leveraging generative AI, the same memory map file was generated using the DTS description in just 1 hour, which included verification and refining the input prompt. This demonstrates a significant reduction in time and effort, and possibly reducing the risk of errors due to the manual generation.

For the presented case of study, the undetected faults from structural tests are analyzed and labeled following the considerations presented in Section III in Fig. 10. For readability reasons, the categories described in Section III as *clock-domain crossing* and *chain configurations crossing* are merged into a single category, *clock-domain and chain crossing*, in all the figures. Fig. 10 shows that the majority of undetected stuck-at faults resides in the ATPG critical class. On the other hand, the number of undetected faults for the Transition Delay fault models is bigger compared to undetected stuck-at, and they spread among different classes.

B. Experimental Setup

The experimental setup in Fig. 11 validates the functional test programs on the manufactured device, referred to as Design under Test (DUT). Afterward, the functional test program is simulated into a logic simulator to produce an extended Value Change Dump (eVCD). The eVCD is used as input stimuli for a commercial fault simulator tool, *Z01X* from Synopsys, for different fault models [33].

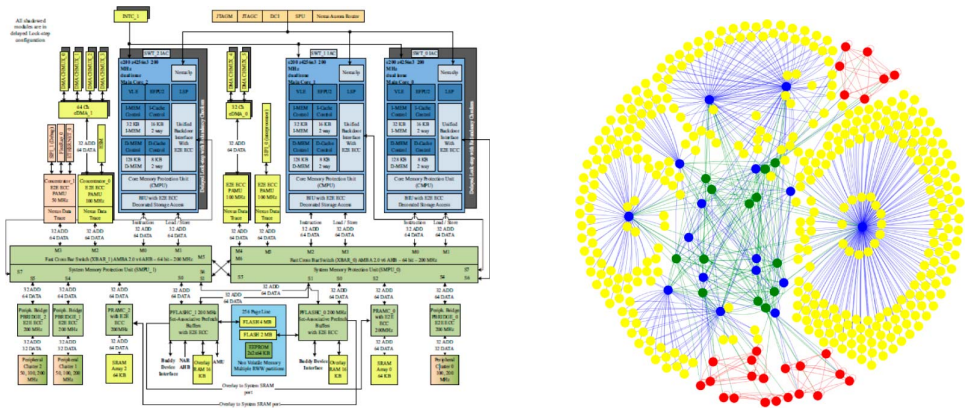
The S^2TL implements the runtime test dispatcher, it parses and executes the generated data structure; it is based on iterative loop over the produced data by the *Grind Nexum algorithm*. Therefore, they can be parallelized during the fault simulation, drastically reducing the computing time. The *Grind Nexum algorithm* can be instructed to split the generated data into multiple partitions to parallelize the fault simulation process. As the last step, fault simulation results are merged to obtain the overall fault coverage of the SLT application, and they are further merged with fault coverage from structural tests (scan patterns and LBIST), about 150k test patterns.

The scan-based fault simulations and analyses to extract the uncovered faults' categories have been performed by using *TestKompress* from the Tessent-ATPG suite.

All the experiments and computations have been executed on a high-performance multi-processor server equipped with a 64-bit 112-core Intel Xeon Gold 6238R CPU at 2.20GHz, 256GB of RAM, a storage system of 10 TBytes, and a Rocky Linux 8 operating system.

C. The Functional SLT Suite

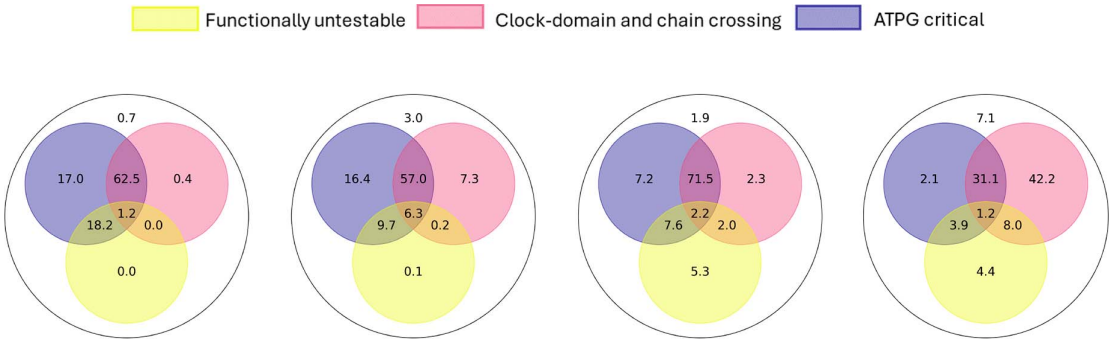
Table I presents the characteristics of the generated SLT functional suite, generated in seconds from the proposed



(a) SoC Architecture with cores (blue), peripherals, crossbars and peripherals bridges (green), and additional crossbar masters and memories (yellow).

(b) SoC Nexum Graph Model with 29 masters (red), 397 slaves (yellow), 15 virtual crossbar masters (green), and 14 virtual crossbar slave (blue).

Fig. 9. A qualitative contrast between the SoC architecture from the user manual with the SoC graph model.



(b) Xbar1 SAF (ca. 5k). (c) Xbar1 TDF (ca. 12k). (d) Xbar0 SAF (ca. 8.5k). (e) Xbar0 TDF (ca. 19k).

Fig. 10. Classification of undetected faults from structural tests for different fault models. All the values illustrated are shown in percentages.

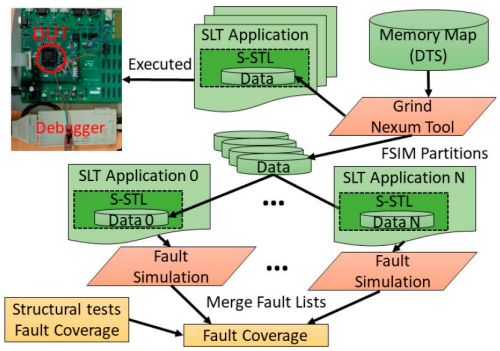


Fig. 11. Experimental setup.

algorithms. It presents for each generated SLT application the visiting algorithms, the number of masters involved (e.g., five masters accounting for three CPUs, two DMA engines, and two cache controllers for instruction and data cache), the total number of generated actions, the execution time and memory footprint. The *Grind Nexum algorithm* is capable of splitting the SLT application into smaller ones, leveraging on the program's regularity. This is done based on the number of total actions and user-defined requirements, generating n-th data structures integrated into the runtime library (S^2TL). The division into smaller parallel and independent SLT applications allows the evaluation of the overall SLT applications in less than 15 days, on average, as the last column in Table I shows (compared to the estimated months). A reduction in the number of transactions in the S^2TL can impact the coverage, as it limits the exploration of functional communication paths between on-chip components. On one hand, applications generated using the node-covering algorithm focus on reaching specific on-chip components. This approach does not exhaustively traverse the graph model but instead generates transactions aimed at covering nodes. As a result, the SLT applications created are smaller and faster, which can reduce test time but may compromise coverage. On the other hand, applications generated using the edge-covering algorithm aim to exhaustively visit all edges in the graph model; for example a direct communication path between a RAM port and Flash exists outside of the crossbar module by means of a DMA. This ensures that all possible functional communication paths between on-chip components are exercised. While this approach can significantly improve fault coverage, it also

TABLE I
GENERATED SLT APPLICATION SUITE AND RTOS CHARACTERISTICS

Name	Visit algorithm	# masters	Tot. # Actions	Execution Time [cc]	Memory Footprint [Kbyte]	Fault Sim. Time [days][1] (est.)	# Fault Sim. Partitions	Avg. Fault Sim. Time per Partition [days]
App1	Node covering from masters	7	3,707	26,481,222	695	129.3	5	12.9
App2	Edge covering from masters	7	6,536	51,071,225	870	249.3	7	9.9
App3	Node covering from slaves	7	2,755	20,279,995	497	99.0	5	3.45
App4	Edge covering from slaves	7	24,051	161,110,659	1,894	786.5	21	11.4
RTOS	NA	NA	4	495,858	164,850	1.5	NA	NA

[1] Stuck-at + Transition Delay fault models for a total of ca. 270k faults.

TABLE II
FAULT COVERAGE FOR STUCK-AT FAULT MODEL (CA. 270K FAULTS)

Test Nature	Test Approach	Stuck-at Fault coverage [%]					
		XBAR1			XBAR0		
		Single	Incr	Δ	Single	Incr	Δ
Structural	Scan-based	94.07	94.07	NA	93.19	93.19	NA
	LBIST	51.99	95.18	1.11	50.44	94.28	1.09
Functional	App1+App2+ App3+App4	49.7	97.08	1.9	53.66	95.95	1.67
	RTOS boot	25.31	97.08	0.0	19.13	95.95	0.00
		Total	**97.08**	**1.9**	**Total**	**95.95**	**1.67**

TABLE III
FAULT COVERAGE FOR TRANSITION DELAY FAULT MODEL (CA. 270K FAULTS)

Test Nature	Test Approach	Transition Delay Fault coverage [%]					
		XBAR1			XBAR0		
		Single	Incr	Δ	Single	Incr	Δ
Structural	Scan-based	87.28	87.28	NA	87.33	87.33	NA
	LBIST	4.08	88.71	1.43	3.15	87.49	0.16
Functional	App1+App2+ App3+App4	29.32	91.11	2.4	31.61	89.76	2.27
	RTOS boot	13.09	91.11	0.0	2.99	89.76	0.0
		Total	**91.11**	**2.4**	**Total**	**89.76**	**2.27**

increases the size and complexity of the SLT applications, leading to longer test times. The RTOS used as the baseline for comparison with the generated SLT suite is a customized version of Micrium µC/OS-III [34] with a software canary in the context switching for maximizing the fault detection. This RTOS is considered a medium-complexity real-time operating system, designed for embedded systems, and mathematical application tasks are added.

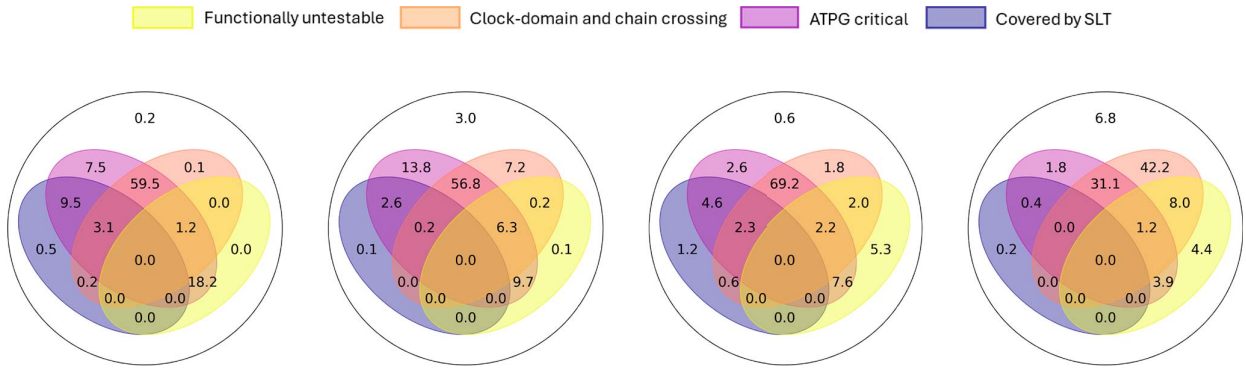
In Table I, the RTOS exhibits a higher memory footprint compared to the SLT application suite due to the inclusion of various kernel modules and complex data structures. However, in terms of execution time, the RTOS shows very low clock cycles and fault simulation time compared to the SLT suite. This is primarily because the measurement is limited to the RTOS bootstrap process, which involves initializing tasks, configuring a subset of peripherals required by the RTOS, and bootstrapping the tasks themselves.

D. Effects on the Crossbar Logic

The functional programs within the proposed SLT suite have been simulated for the Stuck-at fault model, and experimental results are presented in Table II, in which the column "Single" represents the coverage of a single test approach, the column "Incr" represents the cumulative coverage of a given test approach with the previous approaches, and the column "delta" (Δ) represents the increment for a given test approach with respect to the previous one. Moreover, ATPG untestable identified faults have been added in the fault coverage of structural test patterns; meanwhile functionally untestable faults following the approach identified in [32] are added to functional approaches.

The Stuck-at fault coverage achieved by the structural tests (scan-based and LBIST) reaches 95.18% for crossbar 1 (XBAR1) and 94.28% for crossbar 0 (XBAR0). Combining functional programs in the SLT suite with visiting algorithms provides additional fault coverage for high results. It adds up to 97.08% for crossbar 1 (XBAR1) and 95.95% for crossbar 0 (XBAR0).

The proposed SLT suite has been fault simulated also for the Transition Delay fault model, and experimental results are presented in Table III. The Transition Delay fault coverage achieved by the structural tests (scan-based and LBIST) reaches 88.71% for crossbar 1 and 87.49% for crossbar 0. Combining functional programs in the SLT suite with visiting algorithms provides additional fault coverage for high results. It adds up to 91.11 % for crossbar 1 (XBAR1) and 89.76% for crossbar 0 (XBAR0). Measuring the complete execution of an RTOS is impractical in terms of fault simulation time, as the RTOS operates in an infinite event-driven loop. Consequently, only the RTOS bootstrapping process and tasks initialization are measured. This approach impacts execution time compared to the SLT suite application, as the RTOS initializes fewer peripherals and performs fewer operations during its bootstrap phase. However, it negatively affects the fault coverage achieved by the RTOS in absolute terms. This is because the RTOS only initializes a limited subset of the available SoC peripherals, and its firmware must be manually written, unlike the proposed SLT methodology, which exhaustively and automatically utilizes all available SoC components. As a result, the increment of fault coverage obtained during the RTOS bootstrap compared to structural tests is a subset of the fault coverage achieved by the SLT suite, ca. 0.37% for SAF, and ca. 0.48% for TDF in crossbar 1; and ca. 1.41% for SAF, and ca. 0.00% for TDF in crossbar 0;



(b) Xbar1 SAF (ca 4k). (c) Xbar1 TDF (ca. 11k). (d) Xbar0 SAF (ca. 7.5k). (e) Xbar0 TDF (ca. 18k).

Fig. 12. Classification of undetected faults from structural tests and SLT covered faults. All the values illustrated are shown in percentages.

the proposed SLT suite results into a more comprehensive and scalable testing approach compared to the RTOS boot.

Additionally, Fig. 12 shows the classification of undetected faults from structural tests and the percentage for each labeled class of faults detected by the SLT suite. On one hand, some faults detected by SLT are still unclassified; on the other hand, other faults detected by SLT resides in different classes (i.e., structural pattern weaknesses).

The total increase of fault coverage for the TDF model is more significant than the increase for the SAF model. The difference is because scan-based test patterns for modules between different cross-domain clocks are hard to detect [6], and they would require a big structural pattern set. Instead, experimental results show how SLT is capable of detecting such faults in a functional and in-field-like manner without adding additional structural test patterns, and it depicts that the presence of functionally untestable faults is considerable, as Fig. 12 shows.

E. Effects on the Un-Core Logic

The effects of the SLT suite are presented in order to further prove its effectiveness on a small subset on-chip components. It is important to highlight that in order to have fault simulations in a reasonable amount of time the chosen fault lists are the undetected faults (ND), or residual faults, from structural test patterns. The undetected fault lists are obtained by removing the ATPG untestable and functionally untestable faults. In Table IV the effects of SLT suite on a CPU, a peripheral bridge, a CAN peripheral and the top-level entity containing all the memory collars for all the memory controllers (outside any LBIST partitions, the MBIST is not considered for structural tests since it focuses on testing the memory array) are presented for the SAF model.

The column "ND covered" is representing the percentage of undetected faults by structural test patterns covered uniquely by the hereby proposed SLT suite, compared with the boot of an RTOS. The last column reports the final fault coverage of structural tests plus SLT suite.

In Table V the effects of SLT suite are presented on the same modules of Table IV for the TDF model.

TABLE IV
EFFECTS OF SLT SUITE ON DIFFERENT ON-CHIP MODULES FOR SAF MODEL

Module	Structural Coverage [%]	# faults	ND faults	ND covered [%]		Final Fault coverage [%]
				RTOS boot	SLT	
CPU	98.03	1.5M	ca. 29k	6.46	13.96	98.60
Periph. Bridge	90.88	76k	ca. 6.7k	6.94	11.60	91.84
CAN Periph.	95.73	435k	ca. 19k	3.97	3.98	96.33
Memory Controllers	94.51	1.4M	ca. 78.8k	2.60	19.38	95.45

TABLE V
EFFECTS OF SLT SUITE ON DIFFERENT ON-CHIP MODULES FOR TDF MODEL

Module	Structural Coverage [%]	# faults	ND faults	ND covered [%]		Final Fault coverage [%]
				RTOS boot	SLT	
CPU	95.13	1.5M	ca. 29k	0.28	1.96	95.56
Periph. Bridge	68.67	76k	ca. 6.7k	0.09	0.64	70.12
CAN Periph.	87.18	435k	ca. 19k	0.00	0.20	87.62
Memory Controllers	84.23	1.4M	ca. 226.5k	0.14	2.68	84.58

A further investigation in the hierarchy of modules in Table IV and Table V showed the majority of the detected faults by the SLT suite resides in the interfaces of the un-core logic to/from the crossbars.

Due to limitations in peripheral configuration and SoC utilization, the RTOS exhibits low fault coverage for undetected faults compared to the proposed SLT suite. While the fault detection capabilities of the RTOS could improve with better SoC utilization, this would require significant manual effort, increasing development time and complexity. In contrast, the proposed methodology automatically generates a comprehensive SLT suite that thoroughly tests all SoC components, offering broader fault coverage without manual intervention. This underscores the efficiency and scalability of the proposed approach over the manual efforts needed to enhance RTOS fault detection.

V. CONCLUSION

The proposed work focuses on automatically generating a functional SLT application for on-chip interaction between components, providing a flexible automated generation methodology of System-Level functional test programs leveraging the abstracted graph-based model of the SoC modeling means of communication (as edges) between on-chip components (as

nodes), and a method for classifying structural test escapes and assessing the capabilities of the SLT suite. Utilizing standardized DTS files streamlines the generation of memory map files with linear complexity relative to the number of on-chip components, ensuring accurate representation of even complex SoCs with minimal manual effort. Graph generation and traversal complete in seconds, making the proposed approach scalable and practical even for SoCs with thousands of components.

Experimental results obtained on a large automotive SoC by STMicroelectronics demonstrate the effectiveness of the proposed generation methodology by detecting, primarily, test escapes from structural test patterns on well-known fault models (SAF and TDF) compared to the RTOS bootstrap.

REFERENCES

[1] H. H. Chen, "Beyond structural test, the rising need for system-level test," in *Proc. IEEE Int. Symp. VLSI Des. Automat. Test (VLSI-DAT)*, 2018, pp. 1–4.

[2] D. K. R. Tipparthi and K. K. Kumar, "Concurrent system level test (CSLT) methodology for complex system-on-chip," in *Proc. IEEE 6th Electron. Packaging Technol. Conf. (EPTC)*, 2014, pp. 196–199.

[3] G. Iaria, P. Bernardi, C. Bertani, L. Cardone, G. Garozzo, and V. Tancorre, "A comprehensive scan test cost model to optimize the production of very large SoCs," *IEEE Trans. Comput.*, vol. 74, no. 4, pp. 1278–1292, Apr. 2025.

[4] G. Iaria, "A novel pattern selection algorithm to reduce the test cost of large automotive systems-on-chip," in *Proc. IEEE 23rd Latin Amer. Test Symp. (LATS)*, Sep. 2022, pp. 1–6.

[5] D. Tille et al., "Towards an automated flow for implementation of dedicated LBIST scan chains for functional safety," TUZ, 2020.

[6] N. Karimi, K. Chakrabarty, P. Gupta, and S. Patil, "Test generation for clock-domain crossing faults in integrated circuits," in *Proc. IEEE Des., Automat. Test Eur. Conf. Exhibit. (DATE)*, 2012, pp. 406–411.

[7] W. K. Al-Assadi, M. V. Joshi, and G. M. Chaudhry, "A BIST technique for configurable nanofabric arrays," in *Proc. IEEE Int. Workshop Des. Test Nano Devices, Circuits Syst.*, 2008, pp. 63–66.

[8] I. Polian et al., "Exploring the mysteries of system-level test," in *Proc. IEEE 29th Asian Test Symp. (ATS)*, 2020, pp. 1–6.

[9] F. Angione et al., "An optimized burn-in stress flow targeting interconnections logic to embedded memories in automotive systems-on-chip," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2022, pp. 1–6.

[10] Y. Li et al., "Concurrent autonomous self-test for uncore components in system-on-chips," in *Proc. VLSI Test Symp. (VTS),* 2010, pp. 232–237.

[11] M. Campbell, "Plenary presentations: Keynote: The product complexity and test—How product complexity impacts test industry," in *Proc. 15th IEEE Eur. Test Symp.*, 2010, pp. 9–9.

[12] *IEEE Draft Standard Test Access Port and Boundary Scan Architecture*, IEEE P1149.1/D2012.e27, Sep. 2012, pp. 1–434.

[13] C. He and Y. Yu, "Wafer level stress: Enabling zero defect quality for automotive microcontrollers without package burn-in," in *Proc. IEEE Int. Test Conf. (ITC)*, 2020, pp. 1–10.

[14] A. Benso, A. Bosio, S. D. carlo, G. D. Natale, and P. Prinetto, "ATPG for dynamic burn-in test in full-scan circuits," in *Proc. IEEE 15th Asian Test Symp. (ATS)*, 2006, pp. 75–82.

[15] F. Angione et al., "A toolchain to quantify burn-in stress effectiveness on large automotive system-on-chips," *IEEE Access*, vol. 11, pp. 105655–105676, 2023.

[16] D. Appello, H. H. Chen, M. Sauer, I. Polian, P. Bernardi, and M. S. Reorda, "System-level test: State of the art and challenges," in *Proc. IEEE Int. Symp. On-Line Testing Robust System Des. (IOLTS),* 2021, pp. 1–7.

[17] S. Biswas and B. Cory, "An industrial study of system-level test," *IEEE Des. Test Comput.*, vol. 29, no. 1, pp. 19–27, Feb. 2012.

[18] D. Schwachhofer, M. Betka, S. Becker, S. Wagner, M. Sauer, and I. Polian, "Automating greybox system-level test generation," in *Proc. IEEE Eur. Test Symp. (ETS),* 2023, pp. 1–4.

[19] D. Schwachhofer, "Optimizing system-level test program generation via genetic programming," in *Proc. IEEE Eur. Test Symp. (ETS),* 2024, pp. 1–4.

[20] D. Schwachhofer, "Large language model-based optimization for system-level test program generation," in *Proc. IEEE Int. Symp. Defect Fault Tolerance in VLSI Nanotechnol. Syst. (DFT),* 2024, pp. 1–6.

[21] A. Apostolakis, M. Psarakis, D. Gizopoulos, and A. Paschalis, "Functional processor-based testing of communication peripherals in systems-on-chip," *IEEE Trans. Very Large Scale Integr.*, vol. 15, no. 8, pp. 971–975, Aug. 2007.

[22] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, and M. S. Reorda, "Test program generation for communication peripherals in processor-based SoC devices," *IEEE Des. Test Comput.*, vol. 26, no. 2, pp. 52–63, Mar./Apr. 2009.

[23] R. Cantoro, S. Sartoni, and M. S. Reorda, "In-field functional test of CAN bus controllers," in *Proc. IEEE 38th VLSI Test Symp. (VTS),* 2020, pp. 1–6.

[24] F. Angione, P. Bernardi, N. d. G. Giardino, G. Filipponi, C. Bertani, and V. Tancorre, "A system-level test methodology for communication peripherals in system-on-chips," *IEEE Trans. Comput.*, vol. 74, no. 2, pp. 731–739, Feb. 2025.

[25] M. Fujita, N. Taguchi, K. Iwata, and A. Mishchenko, "Incremental ATPG methods for multiple faults under multiple fault models," in *Proc. 16th Int. Symp. Quality Electron. Des. (ISQED)*, 2015, pp. 177–180.

[26] C. Hobeika, C. Thibeault, and J. F. Boland, "Illegal state extraction from register transfer level," in *Proc. 8th IEEE Int. NEWCAS Conf.*, 2010, pp. 245–248.

[27] I. Pomeranz, "Built-in generation of functional broadside tests using a fixed hardware structure," *IEEE Trans. Very Large Scale Integr.*, vol. 21, no. 1, pp. 124–132, Jan. 2013.

[28] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* New York, NY, USA: Springer Publishing Company, 2013.

[29] Fujiwara and Shimono, "On the acceleration of test generation algorithms," *IEEE Trans. Comput.*, vol. C-32, no. 12, pp. 1137–1144, Dec. 1983.

[30] *Devicetree Specification - Release v0*, vol. 3. Accessed: Oct. 20, 2023. [Online]. Available: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.3/devicetree-specification-v0.3.pdf

[31] P. Bernardi et al., "On-line software-based self-test of the address calculation unit in RISC processors," in *Proc. IEEE Eur. Test Symp. (ETS)*, 2012.

[32] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *Proc. IEEE Des., Automat. Test Eur. Conf. Exhibit. (DATE)*, 2013, pp. 1462–1467.

[33] P. Bernardi, "Fault grading of software-based self-test procedures for dependable automotive applications," in *Proc. IEEE Des., Automat. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2011, pp. 1–2.

[34] J. J. Labrosse, *UC/OS-III, The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs.* Weston, FL, USA: Micrium Press, 2009.

Francesco Angione (Member, IEEE) received the M.Sc. degree in embedded systems track and the Ph.D. degree in system-level-test techniques for automotive SoCs from CAD & Reliability Group, Politecnico di Torino, in 2020, and 2025. He is currently a Researcher in the same institution. His research interests include real-time operating systems, computer architectures, and their dependability.

Paolo Bernardi (Senior Member, IEEE) received the Ph.D. degree in computer science. He is currently an Associate Professor with the Electronic CAD and Reliability Research Group, Politecnico di Torino University. His interests include system-on-chip tests and reliability. He is the Program Chair of the IEEE International Test Conference 2025 and has served as the General Chair of the IEEE European Test Symposium 2023.

Giusy Iaria (Member, IEEE) received the Ph.D. degree in computer and control engineering from the Polytechnic of Turin, Italy, in 2025. She is currently a Research Fellow with the Polytechnic of Turin, where she is part of the CAD & Reliability Research Group, with a strong focus on embedded electronics testing for large automotive devices.

Claudia Bertani received the M.S. degree in electronic engineering from the Politecnico di Milano. She is currently a Product & Test Engineer Manager with STMicroelectronics. She is currently managing the team in charge of system-level test introduction on automotive ADAS system-on-chip products. She has more than 20 years of expertise in the semiconductor industry.

Vincenzo Tancorre received the B.S. degree in electronics engineering from the Politecnico di Bari. He is currently a Yield Enhancement Engineer with the Automotive Group, STMicroelectronics. His research interests include test-related process monitoring using diagnostic solutions for memories and unstructured logic based on design-for-testability methodologies.