A System-Level Test Methodology for Communication Peripherals in System-on-Chips

A System-Level Test Methodology for Communication Peripherals in System-on-Chips

Francesco Angione*, IEEE Student Member, Paolo Bernardi*, IEEE Senior Member,
Nicola di Gruttola Giardino*, IEEE Student Member, Gabriele Filipponi*, IEEE Student Member,
Claudia Bertani†, Vincenzo Tancorre†
*Department of Control and Computer engineering, Politecnico di Torino, Turin, Italy
†STMicroelectronics, Agrate Brianza, Italy

***Abstract*—This paper deals with functional System-Level Test (SLT) for System-on-Chips (SoCs) communication peripherals. The proposed methodology is based on analyzing the potential weaknesses of applied structural tests such as Scan-based. Then, the paper illustrates how to develop a functional SLT programs software suite to address such issues. In case the communication peripheral provides detection/correction features, the methodology proposes the design of a hardware companion module to be added to the Automatic Test Equipment (ATE) to interact with the SoC communication module by purposely corrupting data frames. Experimental results are obtained on an industrial, automotive SoC produced by STMicroelectronics focusing on the Controller Area Network (CAN) communication peripheral and showing the effectiveness of the SLT suite to complement structural tests.**

***Index Terms*—System-Level Test, functional testing, fault simulation, Automotive SoCs.**

I. Introduction

THE manufacturing test flow is in charge of ensuring that SoCs have been manufactured without faults, before shipping products to customers. Due to the increasing complexity of SoCs, a significant amount of faults may be still untested after the manufacturing test flow [1]. Structural tests focus on single-component testing, and they often lack testing component interactions and interactions of communication peripherals with an external driver [1].

In order to comply with increasing quality requirements, such as *ISO26262*, in the last decade, a new step in manufacturing test flow has been added right before the final test. The additional test step called the System-Level Test (SLT), resembles the final application, workload, and environment as much as possible, including external communication. Therefore, it is a functional test step for exercising the SoC as a whole. The intention is to cover the mentioned coverage weaknesses of structural tests for already existing fault models.

This work proposes a methodology for generating an SLT suite of functional test programs for communication peripherals. The major innovative elements of the illustrated generation strategy are the following:

- There exist potential structural testing weaknesses that may arise from structural tests such as Scan-based or Built-In Self-Test (BIST); a preliminary phase of the approach aims at highlighting communication peripheral elements that may suffer from such weaknesses.
- Based on the analysis above, guidelines to create SLT-oriented functional test programs are provided in a deterministic and generic form; pseudo-codes are provided to sketch SLT program flows, while data patterns come from state-of-the-art approaches and are integrated into the final SLT suite [2].
- Guidelines are provided for designing a companion module for communicating with the communication peripheral under test; this is crucial to improve the detection of faults located in detection and correction logic.

Experimental results are presented for the Controller Area Network (CAN) module of an Automotive SoC manufactured by STMicroelectronics. The CAN module includes embedded memory elements, communication paths with the inside and outside of the chip, and has a detection/correction logic for errors in the received and transmitted data. The considered design is ”polluted” by Scan Chains with different clock domains, Logic-BIST (LBIST), and Memory-BIST (MBIST) based Design For Testability (DfT) domains, which potentially contribute at introducing untested logic for structural tests [3].

The CAN module under test counts about 436K for Stuck-At Fault (SAF) and Transition Delay Fault (TDF). Increasing the fault coverage of structural tests over SAF and TDF models is the major purpose of the proposed methodology. The original coverage guaranteed by the manufacturing test suite of structural tests, including Scan, LBIST, and MBIST tests, is 97.89 % for SAF and 89.38 % for TDF. The synthesized SLT functional test procedure provides increments of 1.12 % and 1.51 %, respectively, reaching 99.01% for SAF and 90.89% for TDF.

Section II provides background, and section III illustrates the proposed methodology. Section IV reports experimental results, and section V concludes the paper.

Manuscript received April 19, 2021; revised August 16, 2021.

Francesco Angione, Paolo Bernardi, Nicola di Gruttola Giardino, Gabriele Filipponi are with the Politecnico di Torino, Torino, Italy. E-mail: <name.surname>@ polito.it.

Claudia Bertani, Vincenzo Tancorre are with STMicroelectronics, Agrate Brianza, Italy. E-mail: <name.surname>@ st.com.

II. Background

A. Manufacturing test flow

The rising complexity of integrated circuits strongly affects the testing scenario. Scan-based tests, which are part of the so-called structural tests, have been introduced to reduce the complexity of manufacturing test flow; they allow the automation of test pattern generation for an integrated circuit. For testing a specific components, BIST modules are introduced in the design, and they fall under the umbrella of structural tests. Common BIST examples are the MBIST, in charge of testing the memory arrays, and LBIST, oriented to test a specific portion of logic. The manufacturing test flow is split into phases to discard faulty devices as soon as possible [1]:

- *Wafer Test* checks for the primary electrical functionalities of the chip before cutting it out from the wafer and it is based on the execution of structural test patterns.
- *Package Test* repeats the application of the structural tests after packaging the chip.
- *Burn-In*, mainly for automotive and safety-critical devices, exacerbates latent faults.
- *System-Level Test* has been recently added as an additional test phase for safety-critical and automotive devices [9]. It verifies the correctness of devices by resorting to complex functional programs in an environment close to the operational one.
- Final Test repeats the previous structural test suite, eventually complemented, ATE-wise, with functional SLT tests where possible.

SLT is needed because the Scan- and BIST-based test approaches mainly focus on a single component testing without emulating the final environment of the devices; in other words, they do not exercise on- and off-chip component interactions, nor between hardware and software. As a result, Hardware implemented protocols, such as the CAN and SPI bus, are not fully tested by structural tests. For instance, it is typical to use loopback strategies to minimize the interaction with the ATE. This is producing coverage drops on detection and correction logic [10]. Scan and LBIST may also introduce some untested logic when the architecture is partitioned into islands or domains. Such domains may be activated separately and some glue logic pitched between them may be structurally untestable. SLT provides a better activation stimuli than LBIST, because the device is functionally exercised [1], [11]–[14].

B. State-of-the-art for communication peripherals testing

Since the advent of SoCs, designers have started to pack more and more peripherals in the same die. This trend has increased the complexity of SoCs as well as their peripheral speed and capabilities. As complexity increases, so does the testing effort; thus, different methods have been developed in the literature for effectively testing peripherals. In the scope of the current work, Table I shows a comparison between different test approaches for communication peripherals.

The first commonly used approach for testing communication peripherals is always the Automatic Test Pattern Generator (ATPG), a highly automated engine with high coverage capabilities based on generating Scan-based patterns. However, ATPG-based approaches do not have any online testing capabilities or at-speed testing. Most importantly, Scan-based patterns test the modules without functionally using the communication peripherals, as already underlined in the previous section.

Another partially automated approach is the introduction of BIST in the SoC, paying the overhead of additional area [4], [5]. Although BISTs provide at-speed high coverage capabilities for stuck-at and partially online testing only at the SoC startup, they are area-hungry, and they need to be designed ad-hoc for every communication peripheral present in the SoC, leading to an abnormous area overhead.

As Software-Based Self-Test (SBST) has emerged as an effective technique for online testing of CPU modules [15], the concept has also been applied to system peripherals [6]–[8], [16]. SBSTs applied to system peripherals, for communication or not, have been developed to be transparent to the application for simple protocol-related testing. SBST and SLT reside under the umbrella of functional tests. On one hand, SBSTs are mainly adopted as test strategies for online testing capabilities of modules required by safety standards, with the main objective of reaching very high absolute coverage figures. On the other hand, the primary aim of SLT is the detection of faults that escape from previous manufacturing test phases. SLT is typically a holistic strategy that focuses on generating complex hardware-software interaction between all the system components on and off-chip, complex protocol functions, and

Test Nature	Test Approach	Pros	Cons	Target
Structural	ATPG	Automated High coverage capabilities	No functional interactions, No online testing, Low Speed Testing	All
	BIST [4], [5]	At-speed testing High coverage capabilities Limited online testing (startup)	Area overhead Limited capabilities for TDF No functional interactions	SPI, RS-232
Functional	SBST [6]	At-speed Online testing Deterministic and automated methodologies	Simple protocol operations No off-chip communications	UART, HDLC, ETHERNET
	SBST [7]	At-speed Online testing	No off-chip communications	CAN
	SBST [8]	At-speed Online testing Off-chip communications	Simple protocols No errors injector	UART, PIA
	Proposed SLT methodology	At-speed testing Off-chip communications Complex HW/SW interactions	Manual efforts No Online testing Additional Tester capabilities	Generic

TABLE I: Comparison between different test approaches for communication peripherals testing.

requiring additional ATE capabilities, i.e., the capabilities of driving the communication pins from the ATE side.

III. Proposed Methodology

This work proposes a methodology for generating an effective SLT suite of functional test programs oriented to complement the testing abilities of structural tests specifically for communication peripherals in SoCs.

The proposed methodology advances the state-of-the-art SLT methodology because it does not rely upon holistic assumptions (i.e., making the system boot is a good SLT strategy). To the best of our knowledge, the proposed approach is the first to detail the analysis of potential structural test weaknesses. This analysis guides the development of the SLT suite for communication peripherals, targeting test escapes of structural tests in the shadowed zones of the circuit. The methodology flow is represented in Figure 1.

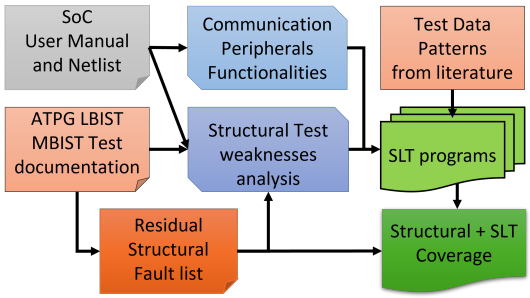


Fig. 1: Flow diagram of the proposed methodology.

Firstly, the potential weaknesses that may arise from structural test flow must be highlighted by qualitatively analyzing the fault list after evaluating ATPG, MBIST, and LBIST approaches. This preliminary analysis step enables the polarization of the successive efforts in the generation of SLT functional programs targeting meaningful circuit regions and faulty conditions.

The functional program generation should cover all the possible working modes and parameters that an in-field application could use, as holistically demanded by SLT. Moreover, the proposed method draws more attention to those functionalities that could be threatened by structural weaknesses.

Stimulating off-chip interactions with the external world, through communication channels exposed by the chip, is a crucial SLT workload for the SoC. Structural tests usually force the communication channels to loop-back configurations, as a consequence, the proposed methodology encompasses the design of a flexible companion module capable of communicating from the ATE side to/from the SoC under SLT, as Figure 2 shows. The companion module becomes indispensable when detection/correction features are available in the peripheral module under test, as they are not going to be tested since loop-back always transports uncorrupted bitstreams.

A. Structural Test Weaknesses Analysis

The analysis of structural test weaknesses is performed manually, with the help of the user manual, the netlist, and

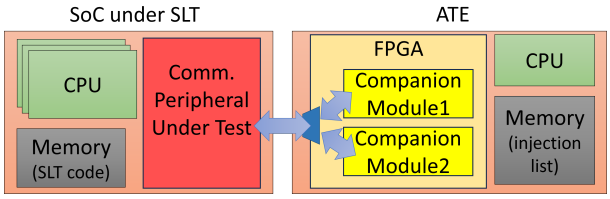


Fig. 2: Companion module view with a companion memory storing error injection information.

the residual structural fault list. The communication module under investigation needs to be stretched as it is generically done in Figure 3. Arrows and labels are added to sub-module boxes that can be extracted easily from the netlist. In the resulting visualization, the colors of the arrows indicate the criticality level of the specific interactions among modules (e.g., dark color is for the most critical and white for low risk), and the alphabetic labels group sub-modules according to their functionality (e.g., under label "A" are classified all functionalities of the internal RAM) represent the following considerations.

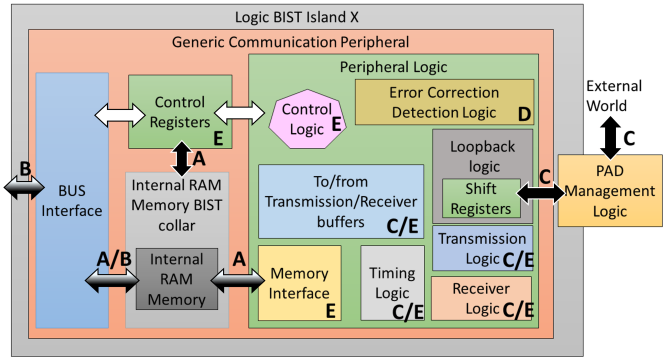


Fig. 3: Detail of a Generic communication peripheral.

What emerges from crossing functionality and structural test limitations is quite intuitive to understand from Figure 3. Therefore, when developing SLT functional programs, meaningful areas to target are the following:

A) **Embedded memory access ports**: they may not be completely covered along structural tests due to collars and memory DfT circuits like MBIST [17].
B) **Interfaces to other on-chip components**: they may be included in different LBIST, or Scan Chain islands can introduce testability issues [3], [11].
C) **Transmission/Reception Interfaces to Chip Top**: Some signals and pins to and from outside the SoC may never be exercised during manufacturing tests.
D) **Detection and correction logic circuits**: they usually include large logic functions resulting in deep circuits that are hard to target by structural tests [1].
E) **Complex hardware-software functions**, like complex protocol functions and synchronization mechanisms, are not exercised [1].

Therefore, an effective SLT suite is a combination of programs capable of systematically tackling all the aforementioned

considerations. The next subsections provides guidelines to punctually address functional programs generation according to the list of critical circuit elements.

B. SLT functional program suite generation

The generation of the functional SLT suite, addressing the identified critical elements, starts from previously established functional methodologies in the state-of-the-art, as reported in Table I. State-of-the-art techniques like [7], [8], [16] cover some of the critical functionalities even using standard test configurations, such as exploiting loop-back configurations from different communication channels and typical frame transmissions/receptions. More efforts are required to complement structural coverage for modules that are not yet very much mentioned in the literature, such as the Error-Management-Logic (EML), Bus-off sequence (if any), shared bus arbitration, and synchronization logic modules of communication peripherals.

The resulting suite, which encompasses known and novel approaches, has been developed manually by a test engineer using the SoC documentation, including structural test application notes, the netlist itself, and the list of residual faults from structural tests. The suggested suite consists of 6 test programs meticulously assembled to specifically target overlooked areas during structural tests. Out of this set of programs, 2 requires the collaboration of the companion module, indispensable to exercise the peripheral by transmitting and receiving packets.

In the following, the functional program specifications are reported in pseudo-code with generic communication peripherals functions that abstract the underlying software.

Moreover regarding the pattern selection, a generic function call $get\_pattern()$ is generically used in the pseudo-code. As such the proposed algorithms can be executed multiple times with various data patterns coming from known literature-based ones such as from [2], where known techniques based on walking bits and checkerboard patterns are illustrated.

The data pattern could also be automatically generated from ATPG-based methodologies [2] or by adopting random methodologies if the cost of fault simulations is not too elevated. In these cases, a generation loop may be set up to refine step by step the pattern set, while a deterministic approach just require a single evaluation.

In the following subsections, the communication software is described from an algorithm perspective, as well as the companion module, for each consideration in Figure 3. All the algorithms presented hereafter produce a signature, which is finally compared with the golden one at the end of every test execution.

1) Embedded Memory access ports: In SoC testing, embedded memory access ports may not be exercised in structural DfT circuits due to MBIST architecture [17]; when the MBIST is operated, it disconnects the memory from the other SoC elements and then it tests only a part of the connections among memory and the rest of the circuit.

Communication peripherals usually provide a dedicated memory to store data. Its functionality can be thoroughly tested by filling the memory with incoming messages. The proposed test must use all transmission/reception buffers if there are more than one. The CPU running the SLT program sends a burst of messages large enough to fill all buffers.

The test program pseudo-code is represented in Algorithm 1.

Algorithm 1 Embedded Memory Access Port test.

```
1:  signature ← 0                                   ▷ Init signature var
2:  while peripheral_rx_buffers_are_full() ≠ True do
3:      data ← get_pattern()
4:      signarure ← signature ⊕ data
5:      send_data(data)
6:  end while
7:  wait_reception()
8:  while peripheral_rx_buffers_are_empty() ≠ True do
9:      data ← receive_data()
10:     signature ← signature ⊕ data
11: end while
```

2) Interfaces to other SoC components: The use of communication interfaces to other SoC components like CPUs, DMAs, etc., is limited during structural tests. They are quite complicated and ATPG resistant because they are composed of multiplexers which are typically difficult to test, especially when a large-sized cross-bar is included in the system. The situation gets even worse if the SoC modules involved in the communications are far from each other in the layout and fall into different LBIST islands [11] or Scan domains [3]. As the structural tests often need to care about power consumption, patterns are not applied to all gates simultaneously, but island per island, or Scan domain by Scan domain.

In addition, complex communication protocols requires coordination with digital/analog circuitry in other domains, leading to coverage loss for structural tests, for example caused by analog IPs being bypassed during Scan tests.

The register interface serves as a collection of registers designed to control the peripheral's behavior, issue commands, store data for transmission/reception, configure additional peripheral units (i.e. timing management logic), and report information from the peripheral controller, including status or interrupt registers. Reading and writing the registers ensure on-chip bus system usage and functional interactions between different LBIST islands. The proposed methodology performs such operations by levering the peripheral under test in an active yet not fully initialized state.

Assuming $\mathbf{R}$ represents the configuration and control peripheral register file, the test can be succinctly expressed in pseudo-code in Algorithm 2.

Many modern micro-controllers offer additional hardware capable of asserting a reset on modules within the entire SoC for performing a clean restart of the communication. Therefore, a functional reset test may be necessary and consists of stopping the peripheral, resetting it, reading back register values, and restarting the peripheral.

3) Transmission/Reception Interfaces to Chip Top: Transmission/Reception Interfaces to Chip Top are crucial for the overall functionality of transmitting and receiving messages to and from outside devices. Structural tests are very limited by the test access, as loop-back strategies are almost always used to reduce the number of pins to contact and control from the

Algorithm 2 Interfaces to other SoC components test.

Require: R, the set of the peripheral register file.
1: $signature \leftarrow 0$ ▷ Init signature var
2: stop_peripheral()
3: assert_functional_reset_peripheral()
4: peripheral_enable_clock() ▷ Avoid init
5: **for each** $r \in \mathcal{R}$ **do**
6: **if** r is init_register **then**
7: $r \leftarrow r \wedge (1 \ll INIT\_field\_pos)$
8: $signature = signature \oplus r$
9: $r \leftarrow r \vee \sim (1 \ll INIT\_field\_pos)$
10: $signature = signature \oplus r$
11: **else**
12: $r \leftarrow 0$
13: $signature \leftarrow signature \oplus r$
14: $r \leftarrow UINT\_MAX$
15: $signature = signature \oplus r$
16: **end if**
17: **end for**

ATE. Therefore, to functionally exercise the Transmission and Receive logic, the Test Algorithm in 3 is used in conjunction with a companion module for sequencing messages (which details are described in Section III-C).

The proposed test program targets the receive/transmission modules by initializing the peripheral to receive/transmit in different configurations. The test program pseudo-code is represented in Algorithm 3.

Algorithm 3 Transmission/Reception to Chip Top test.

Require: B_{tx}, the set of transmission configuration modes.
Require: B_{rx}, the set of reception configuration modes.
Require: $shared$, data shared between ATE and the DUT.
1: $signature \leftarrow 0$ ▷ Init signature var
2: enable_companion_module("$message\_sequencer$")
3: **for each** $btx \in \mathcal{B}_{tx}$ **do**
4: **for each** $brx \in \mathcal{B}_{rx}$ **do**
5: configure_peripheral(btx, brx)
6: **if** is_transmission_enabled(btx) **then**
7: $data \leftarrow$ get_pattern()
8: $signature \leftarrow signature \oplus data$
9: send_data($data$)
10: **else**
11: $signature \leftarrow signature \oplus shared$
12: **end if**
13: wait_reception()
14: $data \leftarrow$ receive_data()
15: $signature \leftarrow signature \oplus data$
16: **end for**
17: **end for**

4) Detection and correction unit: Many peripheral protocols are designed to be susceptible to faulty behaviors. As such, they implement an Error-Management-Logic (EML), which increases reliability and robustness by introducing detection and correction capabilities.

The detection functionality refers to the ability of the peripheral to assess when the transmitted/received data is wrong. It is usually implemented by protection codes inserted into the transmitted data. In addition to detection, correction logic enables the repair of erroneous data using error correction codes transmitted along the data. The correction is limited as it usually tolerates a maximum number of flipped bits.

Moreover, the bus error detection logic ensures that nodes can detect and report anomalies on the bus. This is achieved by identifying incorrect messages and faulty conditions, thus preventing any node from disrupting the bus with faulty outputs. Additionally, malfunctioning nodes can recognize their errors and disconnect themselves from the bus, entering a *bus-off* state until they can re-synchronize and safely rejoin communication.

The test program pseudo-code is represented in Algorithm 4. In this case a companion module capable of injecting errors in message frames is needed. Injection can be done with different strategies as detailed in Section III-C.

Algorithm 4 Detection and correction unit test.

1: $signature \leftarrow 0$ ▷ Init signature var
2: enable_companion_module("$pattern\_recognizer$")
3: $data \leftarrow$ get_pattern()
4: $signature \leftarrow signature \oplus data$
5: send_data($data$)
6: wait_reception_or_errors()
7: **if** peripheral_has_errors() **then**
8: $data \leftarrow$ get_failed_transmitted_data()
9: **else**
10: $data \leftarrow$ received_data()
11: **end if**
12: $signature \leftarrow signature \oplus data$
13: **if** support_bus_off() **then**
14: wait_bus_off()
15: **end if**
16: disable_companion_module()
17: $data \leftarrow$ get_pattern()
18: $signature \leftarrow signature \oplus data$
19: send_data($data$)
20: wait_reception()
21: $data \leftarrow$ receive_data()
22: $signature \leftarrow signature \oplus data$

5) Complex hardware-software functions: The transmission logic, which was already the object of investigation in subsection 3, assumes a pivotal role in handling the transmission of messages, specifically when a priority scheme is part of the peripheral's protocol. When the protocol supports message priority, the transmission handler incorporates complex logic responsible for instigating a so-called *transmission scan* to evaluate pending requests and pinpoint the highest in priority.

The test program pseudo-code is represented in Algorithm 5.

Communication protocols can be based on a dedicated bus, connecting each node to another separately, or on a shared bus between nodes. The shared bus requires additional logic as nodes must implement functionality to perform the arbitration on the shared bus. Indeed, a node starting a transmission can face the cases when the bus is idle (e.g., other nodes are not using the bus) or another node is using the bus. Although, in the first case the only active node will take the bus; bus sensing is not enough when N nodes start transmission simultaneously. Modern protocols introduce an initial phase of transmission called the arbitration phase.

The arbitration logic can be tested by employing the proposed companion module in *message sequencer* mode requiring a shared ID among SLT and ATE. Assuming lower ID has higher priority, the test program pseudo-code is described in

Algorithm 5 Complex transmission functions test.

Require: ID_1, ID_2, ID_3, s.t. $ID_1 > ID_2, ID_3 > ID_2$.
1: $signature \leftarrow 0$ ▷ Init signature var
2: $data \leftarrow$ get_pattern()
3: $signature \leftarrow signature \oplus data$
4: send_data(ID_1, $data$) ▷ Send $data$ with ID_1
5: $data \leftarrow$ get_pattern()
6: send_data(ID_2, $data$) ▷ Send $data$ with ID_2
7: $data \leftarrow$ get_pattern()
8: send_data(ID_3, $data$) ▷ Send $data$ with ID_3
9: cancel_transmission(ID_2)
10: cancel_transmission(ID_3)
11: wait_reception()
12: **if** rx_contains(ID_2) $\vee$ rx_contains(ID_3) **then**
13: ▷ Failure
14: **end if**
15: $data \leftarrow$ receive_data()
16: $signature \leftarrow signature \oplus data$

Algorithm 6, while the *message sequencer* mode is described in Section III-C.

Algorithm 6 Synchronization functions test.

Require: ID, predefined ID between ATE and this program.
1: $signature \leftarrow 0$ ▷ Init signature var
2: enable_companion_module("$message\_sequencer$")
3: $data \leftarrow$ get_pattern()
4: $signature \leftarrow signature \oplus (ID + 1)$
5: send_data(ID, $data$) ▷ Send $data$ with ID
6: ▷ Companion senses a frame and transmits ID+1.
7: $data\_ID \leftarrow$ receive_data_ID()
8: $signature \leftarrow signature \oplus data\_ID$
9: $data \leftarrow$ get_pattern()
10: $signature \leftarrow signature \oplus (ID - 1)$
11: send_data(ID, $data$) ▷ Send $data$ with ID
12: ▷ Companion senses a frame and transmits of ID-1.
13: $data\_ID \leftarrow$ receive_data_ID()
14: $signature \leftarrow signature \oplus data\_ID$

Finally, the timing of communication protocols must be tested. This test requires the implementation of strict synchronization, and the assistance of a companion module able to introduce an arbitrary delay into transmission frames is crucial.

C. Companion module

Different companion modules can be hosted on the ATE FPGA; they can reside on the ATE at the same time with their inputs and outputs multiplexed. Tthe right functionality is activated and communication supplied along the execution of the SLT functional programs that need external support from the ATE.

According to previous subsection III.B, the companion module functionalities needed to complement some of the SLT firmware running on the chip under test are several:

- Message reception and transmission from and to the peripheral under test, also called message sequencer.
- Error injection within the content of specific message segments transmitted to the peripheral under test.
- Delay injection during message transmission.

The message sequencer mode transmits predefined messages to the SoC. Pre-generated messages are stored on-board the companion module in the ATE, and sent out according to the tested communication protocol. The companion module architecture for this mode consists of a central entity and a memory component. The messages can be either hand-crafted, extracted from verification stimulus, or auto-generated pseudo-randomly. Upon receiving an enable signal, i.e., from a digital General Purpose I/O (GPIO), it transmits the message and it returns to an idle state upon completion.

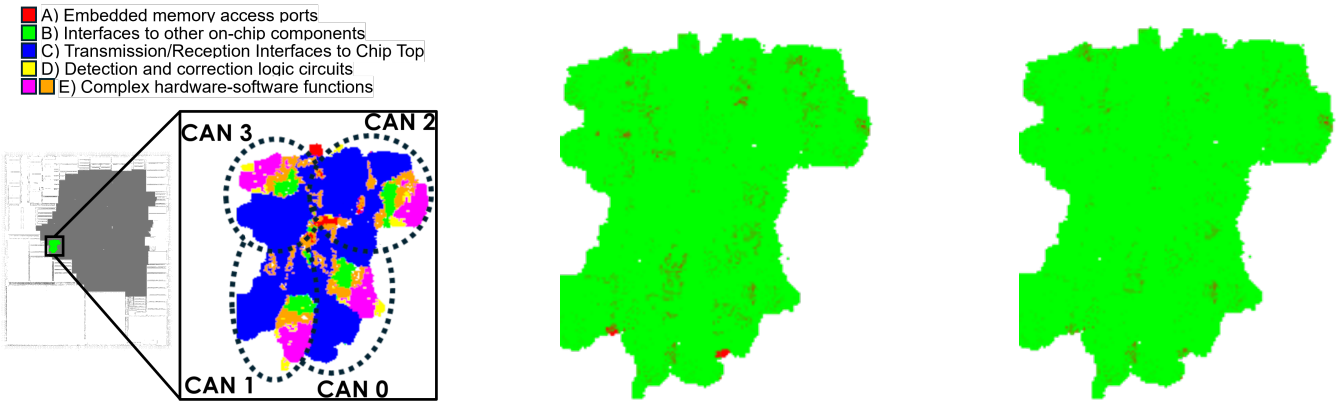
Three possibilities can be explored for error injection: error inject errors according to information already provided before running the test (i.e., a list of times to inject at); a pseudo-random injector that injects errors at random times and locations in the message data frame (i.e., a pseudo-random value determines injection times); a communication protocol injector that knows about the protocol characteristics and corrupts them on purpose (i.e., leaving start and stop bit for too long or short time).

Regarding delay injection, a companion module is required to introduce an arbitrary transmission delay. The introduced delay forces a de-synchronization that exercises the communication peripheral modules aimed at the fine adjustment of the communication timing. Moreover, when the delay introduces a misbehaviour that cannot be corrected, the error detection is exercised as well.

IV. Experimental results

The proposed case study is a cluster of four CAN modules embedded into a 40 nm Automotive SoC manufactured by STMicroelectronics. The SoC has a multicore architecture with three 32-bit cores, it has 6 MB of flash memory and 128 KB of general-purpose SRAM. The SoC design is equipped with multiple LBIST partitions and Scan Chain Domains.
Figure 4 presents a layout heatmap (extracted from the actual physical implementation) for the CAN peripheral in the DUT, where the 4 different controllers can be observed. Figure 4a shows a color-driven layout map of the CAN peripheral logic depending on the critical regions highlighted by the structural test weaknesses analysis, as detailed in section III.A (see the legend in the figure). In this map there are at least two very localized hot spots, located in the error detection/correction logic, and in the Transmission/Reception interface to the chip top, in the southern part of the heat map. This experimental evidence confirms the hypothesis formulated in section III.A about structural test weaknesses. As a preview of the final results fully reported later, Figure 4b provides the layout heatmap of the CAN peripheral, in red the untested SAF faults, and in green the tested ones by all the structural tests. Meanwhile, Figure 4c presents the effect of applying the proposed SLT suite complementing structural tests. Many red spots disappear or the areas colored red are much lighter after SLT.

The algorithms presented in section III.B have been implemented for the cluster of CAN modules. Test patterns are provided according to the deterministic approach shown in [2]. The resulting suite of 6 functional test programs is presented in Table II; it reports, for each program, the targeted weakness, as well as the execution time in terms of clock cycles,



(a) Function-based coloured layout plot.

(b) Structural SAF Coverage heatmap plot.

(c) Structural and SLT suite SAF Coverage heatmap plot.

Fig. 4: Layout view of the four CAN controllers in the CAN peripheral of the DUT.

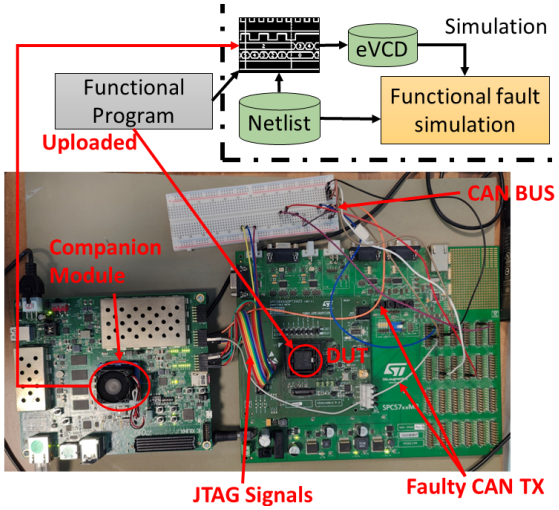


Fig. 5: Laboratory experimental setup.

the memory footprint (code and data), their fault simulation grading time, and their approximate development time by one person for both program development and verification and companion module development and verification. The functional SLT suite requires 11,622,737 clock cycles, which translates into 145 ms with a clock frequency of 80 MHz (the maximum reachable clock for the CAN peripherals in the DUT). The suite was developed in 136 hours or 8.5 days by two test engineers.

Meanwhile, the companion modules are synthesized and implemented in a Xilinx MPSoC ZCU 104 Ultrascale Plus+ equipped with 4 Arm A53 cores running a Linux-based operating system and connected to a host computer through Ethernet. The FPGA is directly controllable by a Linux operating system through the PYNQ framework. The laboratory setup is shown in Figure 5. This allows the connection of an external CAN node to the chip top allowing off-chip by the DUT, including all features related to the error injections. The instantiated companion modules occupy 13,071 LUTS, 6,441 FlipFlops, 2 BRAM cores, and three external I/O pins of the programmable logic; it was synthesized and implemented with a clock of 2 MHz.

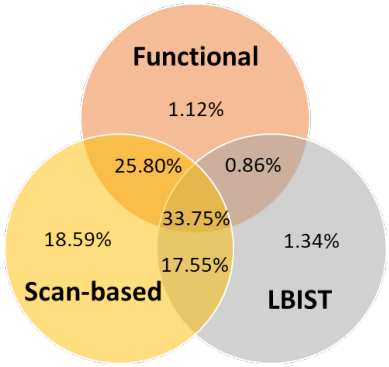
To grade the System-Level Test functional programs, functional fault simulations [18] are performed using a commercial fault simulator *Z01X (Synopsys)*, for Stuck-At-Faults (SAF) and Transition Delay Faults (SDF) fault models. Table III presents SLT fault coverage results including "Single", "Incremental", and "delta" Δ coverages, which represent the individual program coverage, the incremental value to previous tests, and the increment to previous ones, respectively. The fault coverage achieved by the structural tests (Scan-based, LBIST and MBIST) reaches 97.89% for 435,967 SAFs and 89.38% for 435,966 TDF. The proposed SLT permits reaching up to 99.01% for SAFs and 90.89% for TDFs. Figure 6a and Figure 6b show that the functional SLT programs add

Test Name	Algorithm Number	Mode[1]	Targeted Weakness	Execution Time [cc]	Memory Footprint [KB]	Fault Sim. Time[2] [h]	Develop. Time [h]
Embedded Memory Access Port	Algorithm 1	LPBK	**A**	2,522,475	9.96	13.96	20
Interfaces to other SoC components	Algorithm 2	NA	**B**	23,870	12.55	0,13	26
Transmission/Reception to Chip Top	Algorithm 3	LPBK, CMP	**C**	332,096	6.42	1.84	22
Detection and correction unit	Algorithm 4	CMP	**D**	7,294,466	6.82	40.39	34
Complex transmission functions	Algorithm 5	LPBK	**E**	1,002,251	6.92	5.55	10
Synchronization functions	Algorithm 6	CMP	**E**	447,579	6.77	2.48	34
			Total	**11,622,737**	**49.44**	**64.35**	**136**

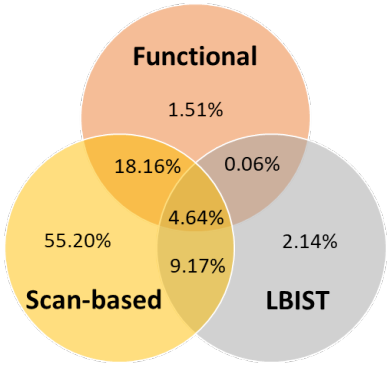
TABLE II: Characteristics of SLT suite for CAN peripheral.
[1] LPBK = Loopback, CMP = Companion Module.
[2] Stuck-at + Transition Delay fault models for a total of ca. 900k faults.

Test Nature	Test Name	Algorithm Number	Fault coverage SAF [%]			Fault coverage TDF [%]		
			Single	Incremental	Δ	Single	Incremental	Δ
Structural	Scan-based	NA	95.69	95.69	NA	87.17	87.17	NA
	LBIST	NA	53.46	97.89	2.2	16	89.38	2.15
	MBIST	NA	1.13	**97.89**	0	0.03	**89.38**	0
Functional	Transmission/Reception to Chip Top	Algorithm 3	40.30	98.45	0.56	12.55	89.65	0.27
	Embedded Memory Access Port	Algorithm 1	41.79	98.65	0.2	12.77	89.73	0.08
	Complex transmission functions	Algorithm 5	25.01	98.66	0.01	5.79	89.74	0.01
	Interfaces to other SoC components	Algorithm 2	13.36	98.96	0.3	5.11	90.63	0.89
	Detection and correction unit	Algorithm 4	39.94	98.99	0.03	11.82	90.67	0.04
	Synchronization functions	Algorithm 6	35.39	99.01	0.02	14.24	90.89	0.22
			Total	**99.01**	**1.12**	**Total**	**90.89**	**1.51**

TABLE III: Fault coverage for Stuck-at fault model (435,967 faults) and Transition delay fault model (435,966 faults).



(a) Stuck-At fault model.



(b) Transition Delay fault model.

Fig. 6: Venn diagrams between Structural, LBIST and functional (SLT) test approaches.

up to 1.12% and 1.51% of fault coverage. LBIST and Scan-based approaches are also quantified in Figure 6, showing that Scan-based approaches cover most of the faults. Functional SLT emerges to be more powerful than LBIST (e.g., covering more faults of the CAN cluster) but LBIST covers more faults "uniquely" than SLT. MBIST adds no unique coverage and is not reported in the diagrams.

The relative improvement by the SLT suite for SAF is about 50% concerning untested faults from structural tests. About TDFs, despite the incremental improvement being higher than for SAFs, the improvement over untested faults is 15%. Given the relatively low coverage for TDF achieved by structural methods, many TDF faults look to be functionally untestable.

V. Conclusions

The proposed methodology aims at filling the gap in structural tests for communication peripherals by providing a functional SLT suite. It illustrates how to create a general recipe for developing an effective functional SLT suite based on the identification of structural tests weaknesses on communication peripheral. The support of companion modules is added to address off-chip communications and error injections. In order to validate the proposed methodology, the CAN peripheral has been selected. However, the methodology is peripheral independent, and portable to other communication modules like UART, SPI, and others.

References

[1] I. Polian *et al.*, "Exploring the Mysteries of System-Level Test," in *IEEE ATS*, 2020, pp. 1–6.
[2] P. Bernardi *et al.*, "On the in-field testing of spare modules in automotive microprocessors," in *IEEE VLSI-SoC*, 2017.
[3] N. Karimi *et al.*, "Test generation for clock-domain crossing faults in integrated circuits," in *IEEE DATE*, 2012.
[4] B. Jose and J. S. Immanuel, "Design of BIST(Built-In-Self-Test)Embedded Master-Slave communication using SPI Protocol," in *ICPSC*, 2021.
[5] S. Saha *et al.*, "Design and implementation of a BIST embedded high speed RS-422 utilized UART over FPGA," in *ICCCNT*, 2013, pp. 1–5.
[6] A. Apostolakis *et al.*, "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices," *IEEE Design & Test of Computers*, vol. 26, no. 2, pp. 52–63, 2009.
[7] F. A. da Silva *et al.*, "A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers," *Electronics*, 2022.
[8] P. Bernardi *et al.*, "Software-Based On-Line Test of Communication Peripherals in Processor-Based Systems for Automotive Applications," in *MTV*, 2006, pp. 3–8.
[9] H. H. Chen, "Beyond structural test, the rising need for System-Level Test," in *VLSI-DAT*, 2018.
[10] P. Aggarwal, "Cost effective manufacturing test using mission mode tests," in *2007 IEEE International Test Conference*, Oct 2007, pp. 1–8.
[11] D. Tille *et al.*, "Towards an Automated Flow for Implementation ofDedicated LBIST Scan Chains for Functional Safety," *TUZ*, 2020.
[12] D. Appello et al., "System-Level Test: State of the Art and Challenges," in *IEEE IOLTS*, 2021.
[13] D. K. R. Tipparthi and K. K. Kumar, "Concurrent System Level Test (CSLT) methodology for complex system-on-chip," in *IEEE EPTC*, 2014.
[14] S. Biswas and B. Cory, "An Industrial Study of System-Level Test," *IEEE Design Test of Computers*, vol. 29, no. 1, pp. 19–27, 2012.
[15] P. Bernardi and et al., "Development Flow for On-Line Core Self-Test of Automotive Microcontrollers," *IEEE Transactions on Computers*, 2016.
[16] M. Grosso *et al.*, "A software-based self-test methodology for system peripherals," in *IEEE ETS*, 2010, pp. 195–200.
[17] F. Angione *et al.*, "An optimized burn-in stress flow targeting interconnections logic to embedded memories in automotive systems-on-chip," in *IEEE ETS*, May 2022, pp. 1–6.
[18] P. Bernardi and et al., "Fault grading of software-based self-test procedures for dependable automotive applications," in *DATE*, March 2011.