POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers

Original
A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers / da Silva, Felipe Augusto; Cantoro, Riccardo; Hamdioui, Said; Sartoni, Sandro; Sauer, Christian; Sonza Reorda, Matteo. - In: ELECTRONICS. - ISSN 2079-9292. - 11:16(2022), p. 2481. [10.3390/electronics11162481]

Availability:
This version is available at: 11583/2970596 since: 2022-08-11T07:15:23Z

Publisher:
MDPI

Published
DOI:10.3390/electronics11162481

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

19 April 2024

electronics

MDPI

Article

A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers

Felipe Augusto da Silva [1,2], Riccardo Cantoro [3], Said Hamdioui [2], Sandro Sartoni [3,*], Christian Sauer [1] and Matteo Sonza Reorda [3]

1 Cadence Design Systems, 85622 Munich, Germany
2 Department of Quantum and Computer Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands
3 Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy
* Correspondence: sandro.sartoni@polito.it; Tel.: +39-320-452-0018

Abstract: In order to match the strict reliability requirements mandated by regulations and standards adopted in the automotive sector, as well as other domains where safety is a major concern, the in-field testing of the most critical devices, including microcontrollers and systems on chip, is a crucial task. Since the controller area network (CAN) bus is widely used in the automotive domain, the corresponding controller ubiquitously appears in all these devices. This paper presents a generic and systematic methodology to develop an effective in-field test procedure for CAN controllers based on a functional approach (i.e., on the adoption of self-test libraries). The method can be customized to match the requirements coming from different scenarios, and allows the test engineer to maximize the achieved fault coverage in terms of structural faults in the different cases. The experimental results we gathered on a representative CAN controller model show that, given two typical testing scenarios, we are able to detect 84.28% and 87.62% of stuck-at faults, respectively, hence demonstrating the effectiveness of the proposed approach.

Keywords: software-based self-test; self-test libraries; online test; automotive electronics; safety

check for updates

Citation: da Silva, F.A.; Cantoro, R.; Hamdioui, S.; Sartoni, S.; Sauer, C.; Sonza Reorda, M. A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers. *Electronics* **2022**, *11*, 2481. https://doi.org/10.3390/electronics11162481

Academic Editor: Xiang Chen

Received: 30 June 2022
Accepted: 1 August 2022
Published: 9 August 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The controller area network (CAN) bus is a communication standard widely used in all those domains where a robust communication connection is required, e.g., in the automotive and industrial sectors. Due to its ubiquitous usage, there is currently a high availability of tools supporting it, easing the integration of standard CAN controllers in many micro controller units (MCUs). For this reason, CAN controller IP cores are also frequently present in automotive system on chip (SoC) devices inside electronic control units (ECUs), with the aim of interfacing and communicating with other ECUs. Due to the nature of this peripheral and its usage, the CAN controller plays a critical role in all applications in which it is adopted, as it enables the communication between different modules within the system. Since failures could severely impact the whole system, special measures should be taken to guarantee the correct functionality of the device. Functional safety (FuSa) standards, such as ISO 26262 for automotive systems, ensure the reliability of such devices by requiring quantitative evaluations of the test effectiveness through fault coverage (FC) computed with respect to permanent structural faults [1]. Throughout the article, we will use the term fault coverage, even though the standard ISO 26262 refers to it as diagnostic coverage (DC). FC is usually evaluated by resorting to well-known fault models, such as the stuck-at-fault model, defining challenging targets to be achieved to match the specified reliability level [2]. To achieve these requirements, testing the CAN controller during its operative lifetime (*in-field test*) is of paramount importance. In most scenarios, in-field testing requires that the testing procedure is conducted on site, without detaching the device under test (DUT) from its system, thus adding extra complexity to the

already hard task of safety verification of the CAN bus controller. Moreover, the in-field test procedure must consider the several constraints coming from both the environment where the core is embedded in, and the application running on it. Due to the aforementioned reasons, there is a high demand for a comprehensive and effective *in-field test* strategy that considers the constraints introduced by the operational environment while still providing a sufficiently high FC.

Two main solutions are generally adopted to test modules, such as the CAN controller in the field, one based on design-for-testability (DfT) techniques, and the other based on the software-based self-test approach [3]. DfT solutions may employ logic built-in self-test (LBIST) and allow an automated and flexible structural test of the DUT, achieving a high FC at the expense of additional on-chip testing hardware. Using DfT for in-field testing requires being able to trigger its execution in an operational environment, which is not always easy to achieve. Testing a device through DfT requires paying special attention to the power consumed while testing the module, as it may exceed that of functional operations. Moreover, DfT test solutions can produce some overtesting, i.e., detecting faults that can never produce any failure, leading to false positives. From a functional safety perspective, those faults are considered *safe* (i.e., their effects cannot impact safety-critical functionalities) and their test does not impact the diagnostic coverage [4]. Lastly, testing the DUT at power-on may not be sufficient, and DfT procedures may require too much time to be executed during the system's operative lifetime (e.g., during the application idle times). Consequently, solutions based on the execution of a self-test library (STL)—a collection of suitably crafted software-based self-test (SBST) procedures—by a CPU are usually adopted, often in combination with other safety mechanisms. In several domains where the functional testing of integrated circuits is required, e.g., the automotive domain, the acronym STL is often used in place of the self-test library. For this reason, throughout this article, we too will use such an acronym. With SBST, it is possible to achieve at-speed testing without using any additional hardware. Developing SBST solutions for CPUs [5–8] and peripherals [9–13] is a well-known topic in the literature. In addition to that, several companies currently provide STLs for their own products [14–20]. Suitable STLs are developed and graded by semiconductor companies and then provided to system companies, which integrate the STL in the application software and devise the most suitable mechanisms for triggering their execution and gathering the results.

In the case of a peripheral core, the idea is to execute a program on the CPU, in charge first of configuring the target core (if required), and then of forcing it to execute some specific transmission operations, checking whether the results match the expected ones. If the test is performed when the device is mounted on a board and the board is part of the final product, no support from any kind of tester is available. Hence, the test is based either on loop-back solutions, where the peripheral core both sends and receives data and then the test program checks whether they match, or it relies on another peripheral core of the same type, possibly hosted on another device connected to the same network. This latter configuration is also relevant to test the wiring of modules. Such wirings might be affected, in the case of industrial installations, by problems related to mechanical *normal operation conditions* conditions, e.g., vibration or electromagnetic noise. In the case that another peripheral is available during the test, the CPU on the second device will also execute a suitable piece of code, performing symmetrical operations. This scheme was investigated and successfully assessed in previous works for simple peripherals [9,11]. When considering serial communication peripherals, it must be noted that the CAN controller is much more complex than peripherals implementing other protocols (e.g., UART, SPI or I2C), as the standard offers a large set of functionalities and configurations that are unique to the CAN bus protocol. The algorithms developed to test such peripherals are thus ineffective and need to be adapted to deal with a higher complexity. The paper [10] is the first work that tackles the in-field testing of CAN controller peripherals using SBST, but it does not take into account the constraints that the environment introduces in terms of testing procedures. This article aims to include those constraints and show how it is possible to adapt the

STLs algorithms proposed in [10] to the new scenario. To the best of our knowledge, this is the first article that tackles the problem of generating the in-field tests of a CAN bus controller, or for peripheral cores of similar complexity, while taking into account the operative scenarios into which they are embedded.

The approach described in this paper tackles the issues related to the peripheral core's size and complexity by presenting a deterministic methodology to test a generic CAN controller with a suitably developed STL. Such a methodology includes a set of systematic strategies to test modules inside the CAN controller, taking into account constraints introduced by the testing environment. Permanent stuck-at faults are targeted, as typically mandated by safety standards, such as ISO26262. When working with the ISO26262 standard, it is standard practice to focus on the stuck-at fault model only. Untestable and safe faults are identified and removed from the total fault list to derive a parameter known as *test coverage* (TC) defined as

$$TC = \frac{DF}{TF - UF - SF}$$

where DF is the number of detected faults, TF is the total number of faults, UF is the number of untestable faults, and SF is the number of safe faults. The UF parameter accounts for *structurally untestable* faults (i.e., faults that cannot be detected due to architectural limitations) and *functionally untestable* faults (i.e., faults that cannot be detected due to functional limitations). We used formal verification techniques to identify those classes of faults, as described in [21]. The main contributions of this work are as follows:

- Identification of some representative scenarios for the in-field test of CAN controllers, to be used for deriving the functional constraints coming from the operational environment.
- Definition of systematic algorithms for the development of an effective STL under the different test scenarios analyzed, corresponding to different choices by the test/safety engineer in terms of safety and cost/intrusiveness of the adopted solution.
- Report of experimental data, such as the memory footprint, execution time, and fault coverage achieved by the test programs in different test scenarios, also taking into account the presence of safe faults.

The gathered results show that traditional unstructured functional solutions can only reach low TC figures, reporting a final 58.42% stuck-at test coverage, while the proposed solution systematically allows much higher TC figures, achieving a final 84.28% and 87.62% stuck-at faults test coverage for two testing scenarios, respectively. These figures are in line with the results achieved with similar SBST-based systematic approaches targeting CPU cores [22] for industrial safety-critical devices, demonstrating that the SBST approach adopted for CPU testing can be effectively extended (following the algorithms described in this paper) also for complex peripheral cores. The paper also proves that a suitably developed functional test can represent (in combination with other safety mechanisms) an effective and flexible solution for the in-field test of CAN controllers.

The rest of the paper is organized as follows: in Section 2, we outline the CAN bus standard, introduce three operative modes, useful when testing, and summarize previous works. In Section 3, we present a generic and systematic approach to develop self-test libraries for CAN controllers, while in Section 4, we discuss the constraints and limitations introduced by the environment in which the tests are conducted. In Section 5, we discuss the case study and the achieved results, and finally in Section 6, we draw the conclusions.

2. Background

2.1. CAN Bus Standard

The CAN BUS protocol [23], released in 1986, is a serial communication standard designed with the aim of making it robust with respect to noisy environments; it was originally developed for the automotive industry. Its robustness is achieved by means of differential signaling, implemented through the *CANH* and *CANL* lines. The synchronization of the receiver to the transmitter is ensured by means of the bit stuffing technique, that allows for a maximal 5 bits sequence of same values. The differential current-based sig-

naling and characteristic impedance terminated transmission lines allow for long-distance transmissions, e.g., 1 km at a low bitrate (50 kbps) or higher bitrates with shorter distances.

This protocol is based on a multi-master bus configuration. The arbitration is achieved by means of employing an *open drain* technology, and relying on a *0 dominant bus* protocol. In this way, in case of a collision due to the simultaneous transmission of a logic 1 from one node while another is transmitting a logic 0, the node that is sending a lower priority message (i.e., a logic 1) is able to recognize such a discrepancy and suspend the transmission.

The CAN standard supports four different frames, i.e., types of message. In the following, we report a brief explanation of these frames:

1. *Data frame*: a message to transfer data from a sending node to one or more receiving nodes.
2. *Remote frame*: a node requests data from a source node. A remote frame is followed by a data frame containing the requested data.
3. *Error frame*: any bus participant may signal an error condition at any time during a transmission.
4. *Overload frame*: a node can request a delay between two data or remote frames.

Messages belonging to the data frame category are divided into different fields. These fields are the *ID* (the identifier of the recipient of the message), *DLC* or data length code (the number of bytes to be sent), *DB* or data bytes (the actual message), and *CRC* for error detection. The recipient notifies the transmitter of the correct reception by means of an acknowledge bit. Data frames come in two formats, with respect to the ID size: 11 bits for the *base* format, and 29 bits for the *extended* format.

Each node generally consists of a controller that elaborates on commands sent from another module (e.g., a microprocessor) and sets everything to correctly send/receive a message. It handles the error conditions as well, providing the outer world with a register-based interface in which any information can be found. A schematic representation of CAN bus configuration, together with the generic implementation of a single node, is shown in Figure 1. The actual implementation of the CAN controller depends on the producer, some of which are SJA1000 by NXP, bxCAN by STMicroelectronics, TI TMS230, and Infineon MultiCan.

Although different, some modules are usually found within any CAN controller implementation. Such modules include the following:

- *Register interface*: a bank of control, status and data registers used as an interface to the controller by the CPU.
- *Timing management logic*: module that handles the timing details of the peripheral.
- *Acceptance filters*: used to check whether the incoming message is intended for the node or it can be discarded.
- *Processing unit*: in charge of managing the transmission/reception operations and configuring the peripheral in its working modes.
- *Error management logic (EML)*: module involved in the management of the various error conditions.
- *Storage FIFO*: used to store received messages.

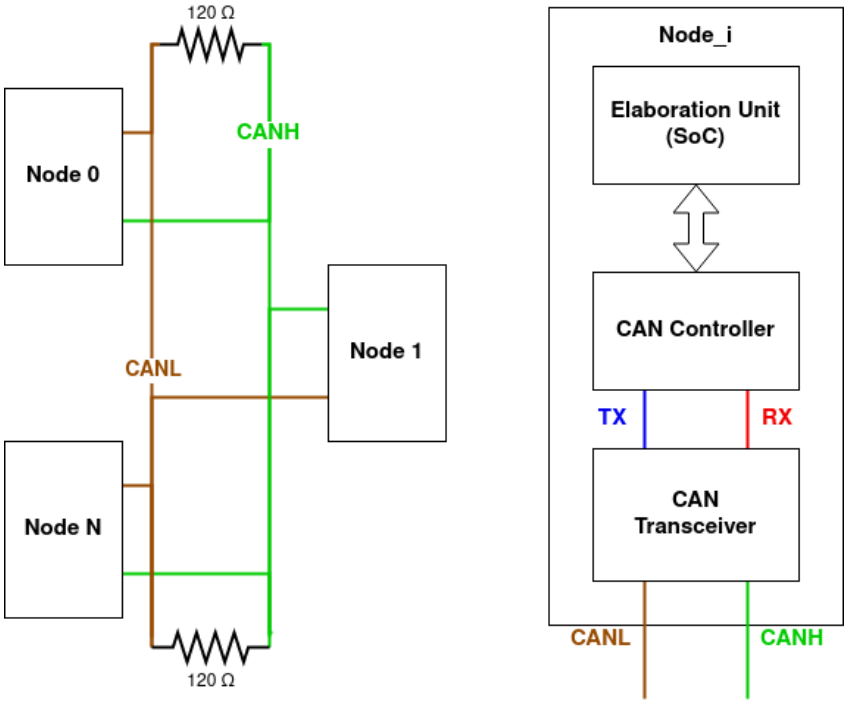

Figure 1. CAN bus configuration.

2.2. CAN Operative Modes

A typical CAN controller supports several working modes, also known as operative modes. In this subsection, CAN operative modes instrumental for testing purposes are described. It is important to notice that not every configuration is introduced here, as some modes, such as stand-by or sleep mode, are not useful when developing an STL. When outlining the operative modes, a distinction between internal TX_{int} and RX_{int} signals and physical TX and RX pins is made. This is done to better clarify how each operative mode configures the transmit-and-receive logic of the CAN controller. The operative modes of interest for this paper are as follows:

1. *Networking mode (NM)*: the configuration used to perform usual transmission and reception of messages among nodes. When in NM, the internal signals TX_{int} and RX_{int} are connected to their respective pins, TX and RX, hence enabling the communication with other nodes. This mode can only be used when there is at least another node configured in NM.
2. *Listen only mode (LOM)*: a configuration where the TX_{int} is physically disconnected from the TX pin. The TX pin is forced to a recessive state, this way the node does not influence the CAN bus as if it was, virtually, disconnected from it. A peripheral configured in LOM can only receive messages from other nodes, as it is unable to start any transmission.
3. *Loopback mode (LM)*: an operative mode in which the node is capable of auto-transmitting messages without sending them on the bus. This is achieved by redirecting the internal signals TX_{int} and RX_{int} so that they are connected, while RX is left floating, and TX is driven with a recessive bit so that it does not influence any ongoing communication.

2.3. Related Works

Several works in the literature tackle the issue of in-field testing of peripherals. The work in [11] describes an approach to generate test programs for peripherals embedded into SoCs based on the adoption of evolutionary tools. This paper introduces an automatic incremental methodology by which test blocks are iteratively generated, so that they collectively obtain the desired fault coverage. This methodology is tested on two peripherals, showing both an improvement in terms of test generation time and final fault coverage.

The two tested peripherals, however, are rather small and do not tackle the problem of how it scales with more complex circuitry. Ref. [9] extends the work in [11], presenting a hybrid approach to developing SBST solutions for peripheral cores. Starting from a set of constraints based on information from the peripheral core, this approach uses an evolutionary core to generate test programs. User-defined constraints are stored into a library, and provide the backbone of the test program and the parameters the evolutionary tool aims at optimizing. This approach is validated on three different peripherals and is capable of achieving significant results on all of them, showing that in-field testing can be a valid solution when testing peripheral cores. This approach, however, does not take into account limitations from the environment into which the peripheral operates, leading to the possibility of generating effective STLs that cannot be launched as is, due to conflicts with other modules embedded in the system. Ref. [12], on the other hand, provides generic algorithms for CPU-based memory testing, highlighting the problems, their solutions and limitations of CPU-based at-speed memory testing. The authors illustrate their approach with examples applied to a RISC microcontroller. Results from this work show that it is possible to increase the final fault coverage while reducing the test time by about 60% the original one. However, since memories are quite different with respect to other peripherals embedded into SoCs, this approach cannot be applied to thoroughly test other modules. Ref. [13] focuses on the trade-offs and benefits that come with reusing manufacturing tests for in-field testing. The testing strategy presented in this paper is described as follows: a programmable component in the electronic control unit, e.g., a microcontroller, sets the device under test into testing mode, accesses the DfT infrastructure of the device under test, applies test vectors through the available DfT hardware and then stores the results for later analysis. Finally, the controller reconfigures the system into a functional mode. This methodology is general, as the device under test is described as a generic ASIC, and the results achieved are high, but it implies the presence of DfT hardware, which adds a significant amount of area and timing overhead to the original circuit. Moreover, testing procedures based on DfT techniques require quite some time to be executed, thus not being the best choice when it comes to in-field testing. The systematic approach proposed by our group [10], finally, introduces a systematic methodology to test CAN controllers on modern SoCs. The presented approach relies on SBST means, showing how to build STLs capable of testing all different functional modes on a generic CAN bus peripheral implementation. The test procedure described in [10] consists of an early on chip stage and system-based communication. Moreover, with such a methodology, diagnostics is possible with multiple nodes. Such a test procedure requires the presence of two nodes, both being able to transmit and receive messages. Experimental results show that the methodology presented in [10] is capable of reaching high fault coverage but does not consider constraints coming from the application run by the system into which they are embedded. In this article, we tackle this shortcoming by proposing a testing methodology based on two different configurations: either the device under test performs a self-testing routine, thus not affecting whatsoever the other nodes on the CAN bus, or the whole network goes into test mode, with some nodes actively partaking into the testing routine, while others do not interact.

3. STL Development Strategies

This section focuses on how to write an effective STL for a generic CAN controller without taking into account the external constraints. The outlined STL strategies are independent from the specific CAN controller implementation, and target all stuck-at faults that are found in the post-synthesis netlist obtained by the CAN HDL description. Usually, the test program is stored in a non-volatile memory in which the application program is also stored. Hence, the test program needs to be as effective as possible with minimal memory space requirements. The experimental results shown later demonstrate that STLs can be written so that their memory size is limited to a few tens of kilobytes.

The proposed approach requires the presence of a test scheduler, i.e., of a hardware- and/or software-based mechanism responsible for triggering the execution of the test program. The identification of the appropriate time to launch the test procedure depends on the operational and safety constraints, and can be flexibly decided by the test engineer. The software in charge of launching the STL execution will also take care of the following:

- Configuring the node under test and, if more nodes take part in the testing routine, other supplementary nodes;
- Informing the system when a fault has been detected.

The test program's internal structure can be thought as a series of sub-modules that target the CAN internal modules presented in Section 2.1. In most cases, the operations performed inside the test program consist of configuring the CAN peripheral, initializing the transmission data structures, issuing transmission and reception operations and monitoring the peripheral status. Data to be used in these operations can either be an optimized set of test arrays, e.g., generated through automated test pattern generator (ATPG), or pseudo-randomly generated by means of software-implemented linear-feedback shift registers (LSFRs). The first approach is usually more effective and sometimes specifically required to test particular hardware properties, e.g., bit stuffing but also more memory hungry than the latter.

Throughout the test procedure, data received from test messages as well as the peripheral's status are acquired; such values are compacted into a signature that is finally compared against the golden circuit's one to check for errors. The signature computation process is a crucial step in the execution of an SBST test procedure, and several articles focus on this topic (e.g., [24–26]). Typically, this is performed by either making use of special hardware structures, e.g., MISR modules within the DUT that can be directly accessed and fed with data through code, or by emulating such hardware in software through arithmetic and logic operations, e.g., by means of sums with carry and xor operations. Previous papers [27–29] showed that when dealing with STLs and suitably implementing/operating the SW-implemented MISRs, the aliasing probability can be reduced to very low values. It is noted that, if enabled, the CAN peripheral can issue interrupts; hence, the test engineer should consider writing appropriate interrupt service routines to handle such cases. This, however, could also lead to a situation where an interrupt never occurs because of the presence of faults. More in general, deviations from the usual execution of the STL may occur, e.g., deadlocks or exceptions from other modules. Works that tackle these issues can be found in the literature (e.g., [22]), providing techniques that make the STL more robust. As an example, to avoid deadlocks, a watchdog timer that brings the system into a safe state in case of infinite waits can be employed. Since in this work we assumed not to be allowed to modify the hardware or add modules, we did not consider this choice that could clearly increase the achieved fault coverage.

Once the STL has been generated, a fault injection mechanism is used to estimate its effectiveness. Fault injections can be carried out in several diverse ways, e.g., by using ad-hoc commercial fault simulation tools, logic simulators, or hardware-based fault injection via FPGA. Regardless of the method, test vectors for the fault injection are obtained from the stimuli applied at the primary inputs of the DUT during the execution of the STL. These vectors are then applied to the synthesized netlist of the DUT where stuck-at faults are injected. In this way, it is possible to assess the achieved test coverage. Such a process is also used as a means of back tracing during the STL generation process. Results obtained during the fault simulation can be arranged so that details on the submodule coverages are obtained, providing insights on what areas of the peripheral need to be better tested (if the achieved fault coverage is not enough) after the basic algorithm proposed for each module has been transformed into test code. In this way, we can also easily identify what portions of the STL need to be improved. Refining the STL can be done in an iterative fashion, with as many cycles of code refinement and fault simulation as required until the desired coverage is achieved. If required, test engineers can finally apply some post-processing techniques to reduce the final STL size by removing redundant portions of code [30–33].

In the remaining part of this section, the sub-modules composing the test program are presented, together with a description of the test algorithm that can be used to detect possible faults inside them. It is noted, however, that testing the *error management logic* module through an SBST approach is not possible, as error conditions are caused by physical phenomenons which cannot be replicated by software means. For this reason, no testing algorithm for this module is provided.

3.1. Storage FIFO Test

The storage FIFO is the module in which data bytes extracted from received messages are stored. In general, this is a large sub-module, accounting for a significant percentage of faults: thoroughly testing the FIFO, hence, is of vital importance. FIFOs can have different implementations, typically in the form of circular buffers or shift registers with pointers to the head and tail of the memory that are updated anytime a read or write operation is performed. Such pointers are used to avoid overflow and underflow conditions and can be accompanied by almost full and almost empty signals to better coordinate memory access. There are, however, some details and issues that are implementation dependent, e.g., the FIFO fill level at which the almost empty/full flags rise. For this reason, specific solutions related to implementation-related problems can easily be added to our test routine. Assuming the FIFO is comprised of n cells, each being m bit wide, a generic algorithm to test it is presented in Algorithm 1.

Algorithm 1: FIFO test algorithm

```
Data: (n, m) where n is the number of FIFO cells and m is the size of each FIFO cell
begin
    /* Fill up the FIFO with 0101-like patterns and generate overrun      */
    write m alternating 0 and 1 bits for all n cells;
    read overrun flag;
    /* Read FIFO content and empty it                                     */
    read content of all n cells;
    read empty flag;
    /* Fill up the FIFO with 1010-like patterns                           */
    write m alternating 1 and 0 bits for all n cells;
    /* Read FIFO content and empty it                                     */
    read content of all n cells;
end
```

These patterns were chosen to recreate a checkerboard-like algorithm, suitable for testing, among other fault models, stuck-at faults in embedded memories. Other test algorithms, including march algorithms, can be adopted to better match the possible defects that may arise in the memory implementing the FIFO. The test consists of two bulks of write and read operations. First, we fill the FIFO with one half of the checkerboard patterns, with the effect of generating an overrun condition, corresponding to a full FIFO, followed by the complete emptying of the memory with the check of its relative empty flag. Next, once the full and empty flags have been tested, we proceed with the remaining half of the checkerboard patterns. In case the almost full and almost empty conditions need to be tested, the test engineer can interrupt the bulk writing/reading operation to check on the relative flags, resuming it after.

3.2. Register Interface Test

The register interface is a set of registers used to issue commands, store data to be transmitted as well as received data, configure other peripheral units such as the timing management logic, and report CAN controller's information, such as status or interrupt registers. For this reason, testing the register interface can be done in conjunction with other tests, with the aim of testing this module in an effective and code efficient way. A generic and systematic way to test these registers is defined in Algorithm 2.

The only aspect that the test engineer must bear in mind when using this approach is that some registers are accessible only when the peripheral is configured into some specific working states, e.g., reset state. Implementations of the CAN bus peripheral may differ in this regard, hence test engineers should carefully pick which registers to observe in different portions of code, depending on the controller's specifications. Moreover, not every register can be read from the peripheral, e.g., command registers. Since the aim of this sub-section is to provide an algorithm to test the register interface of the CAN peripheral, the approach here presented and based on the readback of registers is intended to verify the correct functioning of the interface and bus system of the CPU. The correct execution of issued commands is not managed by this submodule and hence can only be deducted by looking at how the peripheral behaves after writing values to these registers rather than by trying to read their values.

Algorithm 2: Register interface test

```
input  : A tuple V:=(C, S, D) of register values, where C is the set of commands to be issued
         by writing to command registers, S is the set of values used to configure the
         peripheral, and D is the set of values to be written to data registers defined for
         every module's testing algorithm ta_i
output : A signature sg derived from acquired register values
begin
    /* Register testing is integrated into other modules testing routines */
    foreach ta_i do
        /* First write registers with required values                      */
        configure input data registers with data d_i;
        configure the peripheral by writing s_i to configuration registers;
        /* Next, read those registers values                               */
        compact data stored into input data registers into sg;
        compact data stored into configuration registers into sg;
        /* Launch operation required from other test routines              */
        write command registers with new configuration data;
        /* Monitor status registers while running test routine             */
        while test routine is running do
            compact status registers value into sg;
        end
        /* Read produced data                                              */
        compact output data registers value into sg;
    end
    return S
end
```

3.3. Incoming Message Filter Test

The incoming message filter is a module that is used by any node receiving data in order to understand whether the message being sent over the bus is intended for that node to be received, and thus to be stored inside the FIFO. Generally speaking, CAN bus peripherals are equipped with an *acceptance mask*, that tells what bits of the incoming message to observe, and an *acceptance filter*, whose value must match with the bits that are to be observed from the incoming message. An incoming message, thus, is accepted if and only if specific fields of the incoming message are configured to have values matching those required from acceptance mask and filter registers. Such fields are the ID, remote transfer request (RTR) and, depending on the peripheral implementation and configuration, the DLC and DB fields. A schematic representation is shown in Figure 2.

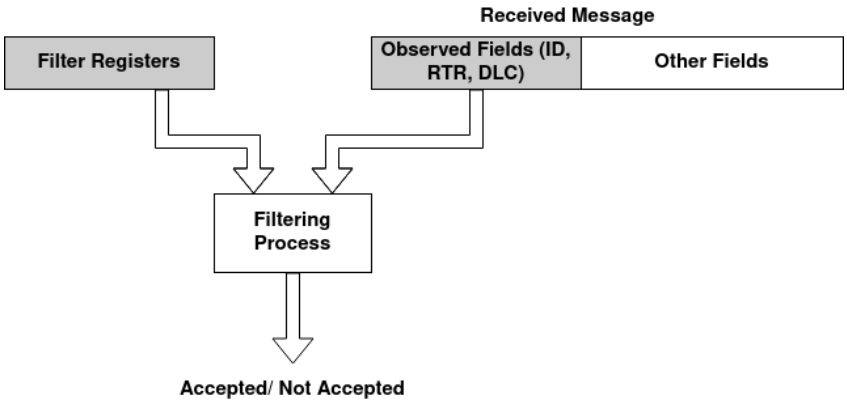

Figure 2. Acceptance filter module.

Hence, testing this module requires configuring these registers and the related incoming message fields with appropriate values, followed by a reception of the active node from the passive node of a message (if the networking mode is employed), or an auto-transmission of a message (if the loopback mode is employed). A generic algorithm to test this circuitry is presented in Algorithm 3.

Algorithm 3: Incoming message filter test algorithm

```
Data: A set A:=(m_i, f_i), where m_i is an acceptance mask configuration and f_i is an
      acceptance filter configuration
Data: A set T:=(t_i), where t_i is a message to be received in accordance to the i-th acceptance
      mask and filter configuration
begin
    /* Test every configuration in A                                          */
    foreach (m_i, f_i) in A do
        set acceptance mask to m_i;
        set acceptance filter to f_i;
        /* Configure the message to be transmitted based on the operative
           mode and check FIFO                                                */
        if loopback mode then
            set message to be auto-transmitted to t_i;
            auto-transmit the message;
        end
        else
            set message to be transmitted from passive node to t_i;
            passive node sends the message;
        end
        check FIFO for a new message;
    end
end
```

With regards to the acceptance filters configurations and transmitted messages fields, the test engineer can either use specific vectors generated by running an ATPG on this module or adopt an iterative pseudo-random approach. The former approach usually yields higher coverage, at the expense of a larger set of configurations to be tested. The latter method, on the other hand, might be useful in case specific configurations should be tested only, without the need of an extensive test.

3.4. Timing Management Logic Test

The timing management logic is the one responsible for configuring the CAN bit timing, which consists not only of the communication bitrate but also the sampling point, i.e., the point of time within any bit at which the bus level is read and interpreted as the

received value. Testing this module can be done by varying the bitrate when performing messages transmission and reception. An effective approach to thoroughly test this module is described in Algorithm 4.

This approach allows to cover most cases in a reasonable amount of time. The aim of this algorithm is not to heavily transmit and receive data, but rather, it is to excite the circuitry related to timing management by initializing it with different configurations and checking its correct functioning by exchanging few messages. For this reason, the parameter n can be set to a small value, even on the order of tens of messages. Looking at frequencies and sampling points, since the node is not broadcasting any message on the bus, the test engineer could decide to pick arbitrary values. In case the CAN bus is intended to work at a specific frequency or a fixed set of frequencies, however, one could test transmission and reception of messages at the predefined frequencies, only.

Algorithm 4: Timing test algorithm

```
Data: A set T:=(br_i, sp_i), where br_i is a communication bitrate and sp_i is a sampling point
Data: An amount n of messages to be exchanged for each configuration
begin
    set the controller in loopback mode;
    foreach (br_i, sp_i) in T do
        configure timing registers based on br_i and sp_i;
        /* Perform auto-transmission operation                    */
        for i ← 1 to n do
            auto-transmit a message;
            read the received data bytes from the FIFO;
        end
    end
end
```

3.5. Processing Unit Test

The processing unit is the sub-module that, given a new issued command or working configuration, is responsible for initializing and configuring other sub-modules within the CAN controller peripheral. It can be conceptualized as a finite state machine that performs its duties in response to commands imparted to the peripheral through instructions or as a response to stimuli coming from other units of the CAN bus controller. Testing such a module, hence, requires to force it into a sequence of specific states so that errors deriving from physical defects can be propagated to primary outputs, thus making them observable. Doing so through an SBST approach implies that such sequences of states have to be generated by means of commands issued to the peripheral. This task may prove to be challenging. One way to face it lies in resorting to formal methodologies, as shown in [6,34]. Such an approach, however, can be computational and time intensive. For this reason, testing this module is commonly achieved as a byproduct of the test of other sub-modules. In fact, working with any of them directly involves the processing unit, as it processes all the required commands and configurations.

This, however, must be done in compliance to the allocated test time and the environment where the test will be conducted, meaning that the amount and type of configurations and issued command must not exceed the time slot reserved for testing purposes and must not disrupt other nodes' activities. Considering the first point, if the time slot is not large enough for all configurations to fit, the test engineer can decide to split them into several sub-routines to be run in several time slots. Concerning the second point, in Section 4, we describe in detail testing execution environments, proposing strategies to perform test operations for each environment.

4. Test Execution Environment

Although an important step, developing an STL is not sufficient in the definition of a test procedure. The CAN controller is connected to a bus on which other nodes may be transmitting or receiving messages: testing the CAN peripheral must not interfere with such operations. For this reason, analyzing the environment in which the test procedure is launched is a crucial step, as it allows to understand which operations can be performed by the test program. Taking the CAN operative modes described in Section 2.2 as a starting point, in this section, we focus on the analysis of the environment in which the test procedure is launched; it allows to understand which of the operations previously defined can be performed by the test program. Taking the CAN operative modes described in Section 2.2 as a starting point, we will describe three representative test scenarios and identify a set of constraints that have to be taken into account when developing the test programs. Each scenario is based on a CAN bus configuration in which there are an *active* node (the node under test that executes its own test program), a *passive* node (a node that, at the occurrence, can assist the active node in its test procedure) and *neutral* nodes not actively involved in the test procedure. Figure 3 shows a bus configuration in which every scenario is employed; the scenarios are explained next.

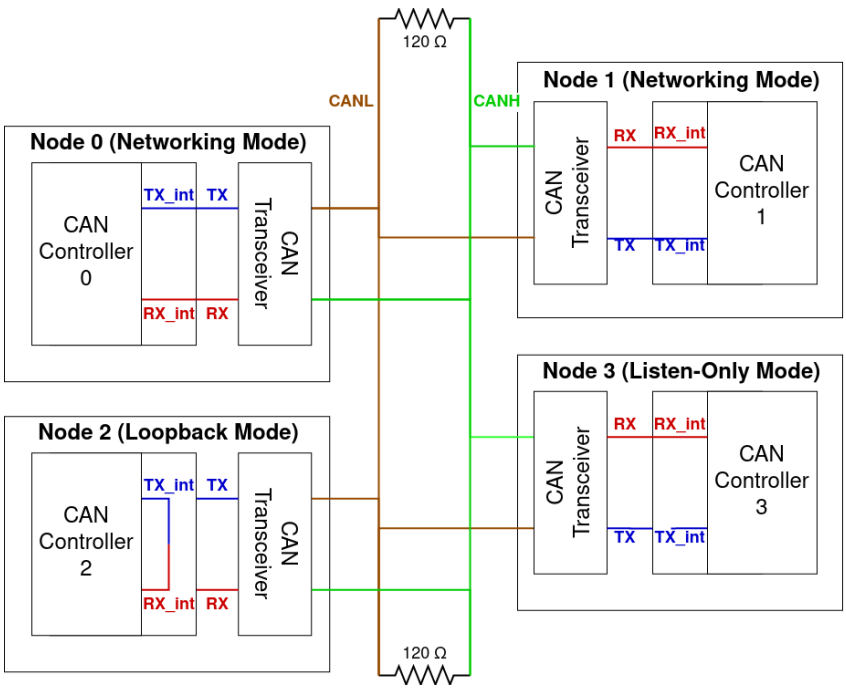

Figure 3. Bus configuration with every test scenario.

4.1. Self-Test Scenario

The self-test scenario refers to the LM, where the active node is capable of self-isolating from the bus in order to auto-transmit messages. This scenario offers the highest flexibility in terms of number of messages to be sent, length of single messages or adopted bitrates, thanks to the fact that the node under test is not interacting with the external bus by disrupting ongoing communications. The test engineer decides how frequently the test program can be launched and can design a rather simple test scheduler that could initiate the testing procedure by means of a timer-generated interrupt. More refined schedulers can also be used but they will not be further investigated, as they are not part of the scope of this article.

Most of the available operations can be executed under this scenario, some of them being tailored exclusively for this situation. Testing the *timing management logic* module as described in Section 3.4, for instance, requires setting the active node to a different bitrate with respect to other nodes, hence leading to the possibility of generating error conditions

in neutral nodes. Other tests, such as the *storage FIFO test*, can also be conducted in different scenarios but are recommended to be run in LM due to the high amount of messages to be transmitted and received.

It is noted, however, that carrying out the test procedure in LM only prevents the use of other working modes. As an example, the circuitry that sets the peripheral in networking or listen-only modes is not used, as well as the circuitry that is responsible for physically transmitting and receiving messages on the bus. In Figure 3, node 2 runs its STL in this scenario.

4.2. Networking Scenario

The networking scenario is the configuration used in the system's operative lifetime. In this scenario, nodes configured in NM communicate with each other by exchanging information based on a predefined higher level protocol, whose implementation depends on the specific system. A generic protocol can work either on a time basis, meaning that communication is cyclic and every node periodically transmits data, or it can work on an event basis, meaning that nodes transmit data whenever specific events occur.

If the protocol is time driven, every node has a given time slot to communicate, and there are idle slots in which no communication occurs. In this case, the STL is developed such that the transmission and reception of test messages are conducted in such idle slots. The test scheduler may initiate the procedure by means of a couple of messages, one at the beginning of the test procedure and one at its end, sent by the active node to configure the passive and the neutral nodes. The idle slot, however, may be short, thus not allowing the transmission of sufficient messages to properly test the node. To overcome this problem, the STL developed in loopback mode, plus an additional transmission and reception of messages in NM, could be used. In this way, even a small amount of messages sent on the bus is sufficient to ensure proper test coverage.

In case of an event-driven protocol, the bus can still be claimed by the active node for test purposes by means of start-of-test and end-of-test messages as previously described. The test procedure is then carried out in a similar fashion, eventually by relying on the mixed loopback and networking approach so as to not occupy too much bus bandwidth. However, due to the asynchronous nature of events, it is not ensured that the test procedure will never collide with an event: for this reason, it is suggested that in this case, the test is carried out using the LM test program. When referring to Figure 3, nodes 0 and 1 run their STLs in the networking scenario.

4.3. Listen Only Scenario

The listen-only scenario is associated to the LOM presented in Section 2.2. A node configured in LOM relies on other nodes to receive data, meaning that the test procedure cannot be run by the node under test itself, as it is unable to transmit any packet. For example, when two nodes are about to use the CAN bus in the networking scenario, the neutral nodes can be configured in LOM so that the test messages sent on the bus can be used to test also the other peripherals.

This scenario, however, introduces several limitations. An STL developed for the listen-only scenario does not test numerous hardware modules, e.g., various working configurations are not excited, the transmitting logic is never exercised and the bitrate is fixed to the one specified in the shared protocol. However, if a CAN controller is connected to a low-end SoC that is constrained in terms of program or data memory size, it may be impossible to store, together with the operative program, a fully developed test program together with its test messages, and it could rely on this mode to carry out a low-cost test, in terms of hardware resources. This scenario is the one adopted by node 3 in Figure 3.

5. Experimental Results

In this section, the experimental setup on which the described methodology has been tested, as well as the associated results, are presented.

5.1. Experimental Setup

The adopted CAN peripheral is an open hardware implementation of the SJA1000 chip by Philips that was embedded into an OpenRisc1200-based SoC. Figure 4 shows a schematic representation of this hardware configuration.

The adopted CAN controller is implemented in a way such that, when the peripheral is set into LM, messages are not only auto-transmitted but also physically sent on the CAN bus. This poses a problem in testing the DUT under the self-test scenario, as the whole test procedure has to be transparent to the other nodes. For this reason, an ad hoc *self-test module* is added to the configuration: when the controller is set into LM, the module physically disconnects the TX_{int} signal from the TX pin while still allowing the auto-transmission mechanism. The devised experimental setup makes use of two nodes, an active node that can either launch the self-test or networking STL, and a second node configured as a passive node in the former case or a neutral node in the latter case. As the whole test procedure is conducted by means of software tools, a simplified version of the CAN bus is adopted. In this version, the physical bus shown in Figure 1 is replaced with a logic-AND of every TX signal, whose output is fed into every node's RX pin: as a consequence, no transceiver is used. Even though this is a simplified implementation, this experimental setup is still representative of how nodes are interconnected in a real system. The fault simulation flow consists of two steps. Initially, we launch a logic simulation where the DUT runs its test program. During this simulation, the CAN controller's input pin values are recorded at each clock cycle. Such values are then used as test vectors in the fault simulation process.

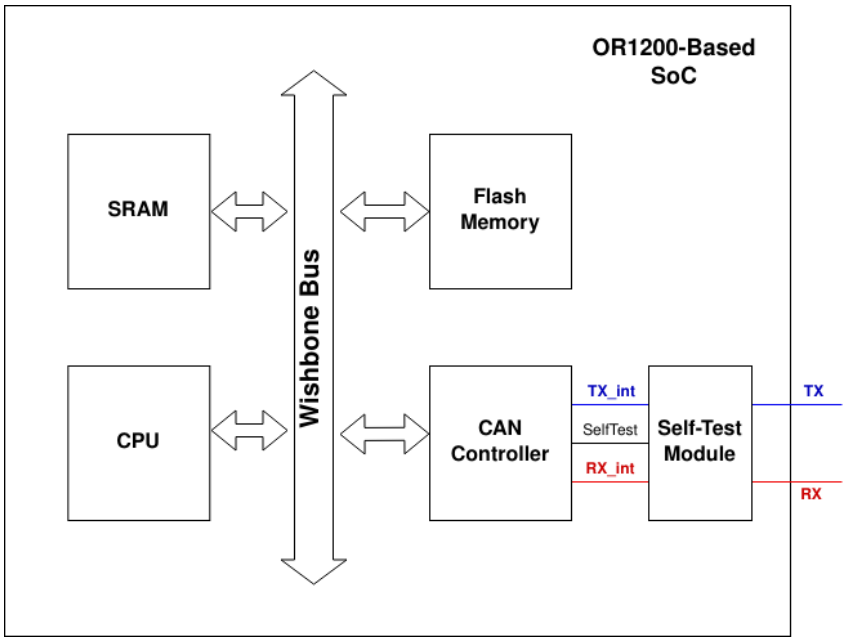

Figure 4. SoC configuration.

We develop one STL for each scenario described in Section 4, i.e., an STL for the self-test scenario, one for the networking scenario, and one for the listen-only scenario. The self-test STL can be thought an independent set of testing routines built as described in Section 3. The networking STL is built on top of the self-test one, with an additional 16 messages sent from the active to the passive node, 16 messages received by the active from the passive one, and 16 messages received by the active node in LOM. The aim of transmitting and receiving messages in the networking and listen-only modes is solely to excite the logic responsible for managing the communication with another node in the given configurations. For this reason, we opted for the aforementioned amount of messages to be exchanged on the bus. This program was developed, assuming that the communication on the bus should not take too much time; the test engineer may eventually

decide to increase the amount of messages. Finally, for the LOM STL, we assume that other nodes exchange test messages on the bus. The STL can hence be thought as two main blocks: the first, which takes care of configuring the correct working mode and bitrate of the peripheral to be tested, and the second, which consists of a series of reading operations of each and every message transmitted on the bus. Some figures regarding the proposed STLs involving test application time (TAT), memory size, fault simulation time, and time required to develop the test routines are reported in Table 1. The TAT has been calculated taking into account a working frequency of 150 MHz for the CPU, a value commonly found in modern SoCs. It is noted that the reported TAT refers to the three test STLs when run as a whole. Given the nature of such STLs; however, the test engineer can split them into a set of smaller subroutines so that the test routine can be run during idle time slots of the peripheral, hence imposing no time overhead. The size of each subroutine, although adjustable, should be such that no testing operation is left unfinished: the largest testing block, however, only requires 2 ms to run.

Table 1. Main figures about the developed STLs.

STL	Test Application Time @150 MHz [ms]	Memory Size [kB]	Simulation Time [Hours]	Development Time [Days]
Self-Test STL	79.7	18	15	12
Networking STL	88.1	20	16	13
LOM STL	33.2	7.5	5	3

Due to the absence of similar works as highlighted in Section 2, in order to assess the effectiveness of our approach we also developed a *comparative test program* that emulates a generic functional program used to launch test procedures whenever no other solution is available. Such a program is written so that active and passive nodes exchange 200 messages in networking mode. The amount of transmitted and received messages was chosen to ensure a sufficiently large number of messages exchanged on the bus. Given that this is a generic functional program, however, all the messages are sent and received in one single working configuration.

All STLs were evaluated by a fault injection mechanism based on a commercial functional fault simulation tool by Synopsys that runs fault-parallel simulations. We performed the fault simulation on the gate-level netlist of the DUT. Test patterns were obtained while performing a logic simulation of the DUT running the STL, recording all input ports of the DUT at each clock cycle. In the same logic simulation, we also recorded the values at the output ports at each clock cycle, thus generating the set of fault-free responses, also referred to as golden responses, that were then provided to the fault simulation tool.

5.2. Simulation Results

Before evaluating the experimental results, some considerations are needed. First, it is necessary to remove safe faults from the active fault list, as they cannot impact the DUT under any functional constraint. It is also necessary to consider faults that affect the error management logic, as they can create false alarms but not functionality failures. Moreover, there is a set of faults that affect specific sections of the peripheral and are mainly due to physical phenomenon, e.g, the re-synchronization of the receiver node due to clock jitters, that require manual analysis due to the constraints of the logical simulation. As it is not possible to reproduce the conditions that excite them through software simulations only, they need to be manually analyzed and verified during the integration tests on the real circuit. Lastly, when running the self-test STL, there are faults associated to a portion of circuitry that cannot be excited, as the test is conducted in LM only. These faults, although not safe in general, can be ignored for the test coverage of this specific operating mode. Safe faults amount to 710 faults; after adding EML-related and manually analyzed faults,

they total 3478 faults. For the self-test STL TC, to the aforementioned faults, we have to add those that cannot be excited under the LM, obtaining a total of 3808 faults.

Table 2 reports the gathered results both for the whole controller and for its sub-modules, namely *registers* (the register interface), *BTL* (bit timing logic, the timing management logic module) and *BSP* (bit stream processor, the processing unit) that also comprises the *ACF* (acceptance filters) and *FIFO* units. As specified in Section 3.5, the test of the processing unit is a byproduct of the test of other modules within the DUT; results are hence reported here for completeness. Columns *networking TC* and *self-test TC* report the test coverage achieved by the relative STLs. In these coverage figures, the 109 faults that were marked as undetected by the fault simulation tool have been removed. Columns *networking AR TC* and *self-test AR TC* report the test coverage of the networking and self-test STLs after removing the previously defined 3478 and 3808 faults, respectively. From this table, it is possible to deduce the following:

- The TC achieved by our methodology—87.62% for the networking TC and 84.62% for the self-test TC—is significantly better than the one achieved by a generic functional program, whose results are showed in the *comparative program TC* column.
- Identifying functionally untestable faults (FUFs) allows to notably enhance the achieved TC, gaining an additional 8% in both cases. It is important to underline that in the real product, other FUFs can typically be identified by taking into account specific characteristics of the applications.
- Results achieved by adopting the LOM strategy highly depend on the number of messages transmitted on the bus. The TC on the left side of the *LOM TC* column is obtained by reading messages that are sent on the bus by the active and passive nodes while executing the proposed networking STL. The TC on the right side of that column can be achieved if every message auto-transmitted by the active node in LM was to be sent on the bus.

As the networking STL is an extension of the self-test one, faults covered by the latter STL are a subset of those covered by the former STL.

Table 2. Achieved stuck-at faults coverage.

Instance Name	#Stuck-at Faults	Networking TC %	Networking AR TC %	Self-Test TC %	Self-Test AR TC %	LOM TC %	Comparative Program TC%
Registers	5352	79.46	83.00	76.65	80.75	17.75–49.54	37.17
BTL	1472	66.37	81.63	64.93	80.36	3.08–56.85	51.58
BSP	31,236	80.15	88.62	76.20	84.94	0.28–64.63	61.95
ACF	1418	52.19	52.41	50.56	51.03	0.00–1.97	3.10
FIFO	17,382	92.88	93.15	91.60	91.88	0.00–86.99	75.11
TOTAL	38,492	79.66	87.62	76.00	84.28	3.48–62.54	58.42

The approach presented in this article is a mix of manual STL development based on ad hoc methods targeting specific modules within the CAN controller and ATPG-based solutions regarding the incoming message filter. For some modules (e.g., the register interface), we described in the paper a new method to test them. The results are comparable to those that are typically achieved on processors and simpler peripherals, with a typical achieved coverage above 80% of faults, proving the effectiveness of the approach [22,35,36]. When comparing the results achieved by this methodology with respect to those obtained in [10], refined with the untestability analysis performed in [21], it is noted that the current approach takes into account the real test environment, while the test programs developed following those approaches was not limited by any test constraint and constitutes an upper bound to the achievable TC. Functional test by means of an STL is not intended to cover every possible fault as it tests the peripheral in specific operating modes: they are usually deployed in conjunction with other safety mechanisms, such as system-level checks,

ECC/parity on internal memories and register banks and CRC/Parity in communication buses. As a further example, to enhance the final TC, a mechanism to continuously monitor the CAN bus could be adopted; moreover, some of the untested faults could also be covered by increasing the number of test messages exchanged on the bus in the networking STL.

6. Conclusions

In this work, we presented a systematic methodology to develop STLs for CAN controllers. The proposed approach takes into account the test scenario into which the STLs are executed, which may change depending on the specific system constraints. Such a methodology allows to achieve consistent results and represents a major improvement with respect to the results achieved by non-systematic approaches, e.g., based on exchanging an arbitrary amount of messages between nodes, that are used when no other solutions are available. We analyzed a few scenarios representative of the different cases that can be found in industrial applications. Results show that STLs developed for the networking and self-test scenarios cover 87.62% and 84.28% of stuck-at faults, respectively, and are comparable to those typically obtained on processor cores and peripherals through SBST means. The listen only program's effectiveness (characterized by a much lower invasiveness) highly depends on how many messages are sent on the bus, and clearly achieves a lower fault coverage. These results are significantly better than those obtainable via any functional test simply operating the CAN controller, amounting to only 58.42% of detected stuck-at faults, hence proving that the proposed method represents a significant advancement in the state of the art in the area. Using the proposed approach, the safety engineer can select the most suitable solution fitting the specific safety targets, operational constraints and costs (e.g., in terms of test time duration and memory footprint).

The proposed approach is particularly suited for all situations where an effective and flexible in-field test of a CAN controller is needed. This method can be easily implemented without any additional cost, as no additional hardware is required. Furthermore, it allows the system's safety matching the requirements set by the most stringent ASIL levels introduced in the ISO26262 standard (possibly in combination with other safety mechanisms) while being compatible with the constraints (e.g., in terms of test duration and memory footprint) of most application environments.

Author Contributions: Conceptualization, R.C., S.S. and M.S.R.; methodology, R.C., S.S. and M.S.R.; software, S.S.; validation, R.C., S.S. and M.S.R.; data curation, R.C., S.S. and M.S.R.; writing—original draft preparation, F.A.d.S., R.C., S.H., S.S., C.S. and M.S.R.; supervision, R.C., S.H. and M.S.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are contained in this article "A Systematic Method to Generate Effective STLs for the In-Field Test of CAN Bus Controllers".

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACF	Acceptance Filter
BTL	Bit Timing Logic
BSP	Bit Stream Processor
CAN	Controller Area Network
DB	Data Bytes
DC	Diagnostic Coverage
DF	Detected Faults
DfT	Design for Testability
DLC	Data Length Code
DUT	Device Under Test

ECU	Electronic Control Unit
EML	Error Management Logic
FC	Fault Coverage
FuSa	Functional Safety
ID	Identifier
LOM	Listen Only Mode
LBIST	Logic Built-In Self-Test
LM	Loopback Mode
MCU	Micro Controller Unit
NM	Networking Mode
SoC	System on Chip
SBST	Software-Based Self-Test
SF	Safe Faults
STL	Self-Test Library
TC	Test Coverage
TF	Total Number of Faults
UF	Untestable Faults

References

1. Chonnad, S.; Iacob, R.; Litovtchenko, V. A Quantitative Approach to SoC Functional Safety Analysis. In Proceedings of the 2018 31st IEEE International System-on-Chip Conference (SOCC), Arlington, VA, USA, 4–7 September 2018; pp. 197–202. [CrossRef]
2. Ross, H.L. *Functional Safety for Road Vehicles: New Challenges and Solutions for E-mobility and Automated Driving*; Springer International Publishing: Berlin/Heidelberg, Germany, 2016. [CrossRef]
3. Psarakis, M.; Gizopoulos, D.; Sanchez, E.; Sonza Reorda, M. Microprocessor Software-Based Self-Testing. *IEEE Des. Test Comput.* **2010**, *27*, 4–19. [CrossRef]
4. Narang, A.; Venu, B.; Khursheed, S.; Harrod, P. An Exploration of Microprocessor Self-Test Optimisation Based on Safe Faults. In Proceedings of the 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Athens, Greece, 6–8 October 2021; pp. 1–6. [CrossRef]
5. Cantoro, R.; Carbonara, S.; Floridia, A.; Sanchez, E.; Sonza Reorda, M.; Mess, J. An analysis of test solutions for COTS-based systems in space applications. In Proceedings of the 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Verona, Italy, 8–10 October 2018; pp. 59–64. [CrossRef]
6. Riefert, A.; Cantoro, R.; Sauer, M.; Sonza Reorda, M.; Becker, B. A Flexible Framework for the Automatic Generation of SBST Programs. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2016**, *24*, 3055–3066. [CrossRef]
7. Cantoro, R.; Foti, D.; Sartoni, S.; Sonza Reorda, M.; Anghel, L.; Portolan, M. New Perspectives on Core In-field Path Delay Test. In Proceedings of the 2020 IEEE International Test Conference ITC, Washington, DC, USA, 1–6 November 2020; pp. 1–5. [CrossRef]
8. Cantoro, R.; Girard, P.; Masante, R.; Sartoni, S.; Sonza Reorda, M.; Virazel, A. Self-Test Libraries Analysis for Pipelined Processors Transition Fault Coverage Improvement. In Proceedings of the 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 28–30 June 2021; pp. 1–4. [CrossRef]
9. Apostolakis, A.; Gizopoulos, D.; Psarakis, M.; Ravotto, D.; Sonza Reorda, M. Test Program Generation for Communication Peripherals in Processor-Based SoC Devices. *IEEE Des. Test Comput.* **2009**, *26*, 52–63. [CrossRef]
10. Cantoro, R.; Sartoni, S.; Sonza Reorda, M. In-field Functional Test of CAN Bus Controllers. In Proceedings of the 2020 IEEE 38th VLSI Test Symposium (VTS), San Diego, CA, USA, 5–8 April 2020; pp. 1–6. [CrossRef]
11. Bolzani, L.; Sanchez, E.; Schillaci, M.; Sonza Reorda, M.; Squillero, G. An Automated Methodology for Cogeneration of Test Blocks for Peripheral Cores. In Proceedings of the 13th IEEE International On-Line Testing Symposium (IOLTS 2007), Heraklion, Greece, 8–11 July 2007; pp. 265–270. [CrossRef]
12. van de Goor, A.; Gaydadjiev, G.; Hamdioui, S. Memory testing with a RISC microcontroller. In Proceedings of the 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), Dresden, Germany, 8–12 March 2010; pp. 214–219. [CrossRef]
13. Cook, A.; Ull, D.; Elm, M.; Wunderlich, H.; Randoll, H.; Döhren, S. Reuse of Structural Volume Test Methods for In-System Testing of Automotive ASICs. In Proceedings of the 2012 IEEE 21st Asian Test Symposium, Niigata, Japan, 19–22 November 2012; pp. 214–219.
14. Hitex. Microcontroller Self-Test Libraries. 2022. Available online: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/ (accessed on 30 July 2022).
15. STMicroelectronics. Guidelines for Obtaining IEC 60335 Class B Certification for Any STM32 Application. 2016. Available online: https://www.st.com/resource/en/application_note/an3307-guidlines-for-obtaining-iec-60335-class-b-for-any-stm32-application-stmicroelectronics.pdf (accessed on 30 July 2022).
16. Cypress Semiconductor. FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library. 2022. Available online: https://www.cypress.com/file/249196/download (accessed on 30 July 2022).
17. Renesas Electronics. SSP Supplemental Add-Ons. 2022. Available online: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html (accessed on 30 July 2022).

18. Microchip Technology Inc. 16-bit CPU Self-Test Library User's Guide. 2022. Available online: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf (accessed on 30 July 2022).
19. ARM. What Is Functional Safety? 2022. Available online: https://www.arm.com/technologies/safety (accessed on 30 July 2022).
20. NXP Semiconductors. S32 SDK for S32K1 Microcontrollers. 2022. Available online: https://www.nxp.com/support/developer-resources/run-time-software/s32-sdk/s32-sdk-for-s32k1-microcontrollers:S32SDK-ARMK1 (accessed on 30 July 2022).
21. da Silva, F.A.; Bagbaba, A.C.; Sartoni, S.; Cantoro, R.; Sonza Reorda, M.; Hamdioui, S.; Sauer, C. Determined-Safe Faults Identification: A step towards ISO26262 hardware compliant designs. In Proceedings of the 2020 IEEE European Test Symposium (ETS), Tallinn, Estonia, 25–29 May 2020; pp. 1–6.
22. Bernardi, P.; Cantoro, R.; De Luca, S.; Sánchez, E.; Sansonetti, A. Development Flow for On-Line Core Self-Test of Automotive Microcontrollers. *IEEE Trans. Comput.* **2016**, *65*, 744–754. [CrossRef]
23. Robert Bosch GmbH. CAN Specification, Version 2.0. 1991. Available online: https://www.kvaser.com/software/7330130980914/V1/can2spec.pdf (accessed on 30 July 2022).
24. Bernardi, P.; Cantoro, R.; Ciganda, L.; Sanchez, E.; Reorda, M.S.; De Luca, S.; Meregalli, R.; Sansonetti, A. On the in-field functional testing of decode units in pipelined RISC processors. In Proceedings of the 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam, The Netherlands, 1–3 October 2014; pp. 299–304. [CrossRef]
25. Lin, B.H.; Shieh, S.H.; Wu, C.W. A fast signature computation algorithm for LFSR and MISR. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2000**, *19*, 1031–1040. [CrossRef]
26. Shenoy, J.; Ockunzzi, K.; Singh, V.; Kamal, K. On-chip MISR Compaction Technique to Reduce Diagnostic Effort and Test Time. In Proceedings of the 2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID), Delhi, India, 5–9 January 2019; pp. 106–111. [CrossRef]
27. Iwasaki, K.; Feng, S.P.; Fujiwara, T.; Kasami, T. Comparison of aliasing probability for multiple MISRs and M-stage MISRs with m inputs. In Proceedings of the Pacific Rim International Symposium on Fault Tolerant Systems, Kawasaki, Japan, 26–27 September 1991; pp. 90–95. [CrossRef]
28. Lu, T.H.; Chen, C.H.; Lee, K.J. Effective Hybrid Test Program Development for Software-Based Self-Testing of Pipeline Processor Cores. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2011**, *19*, 516–520. [CrossRef]
29. Perez Acle, J.; Cantoro, R.; Sanchez, E.; Sonza Reorda, M.; Squillero, G. Observability solutions for in-field functional test of processor-based systems: A survey and quantitative test case evaluation. *Microprocess. Microsyst.* **2016**, *47*, 392–403. [CrossRef]
30. Touati, A.; Bosio, A.; Girard, P.; Virazel, A.; Bernardi, P.; Reorda, M.S. An effective approach for functional test programs compaction. In Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, Slovakia, 20–22 April 2016; pp. 1–6. [CrossRef]
31. Guerrero-Balaguera, J.D.; Rodriguez Condia, J.E.; Reorda, M.S. A Novel Compaction Approach for SBST Test Programs. In Proceedings of the 2021 IEEE 30th Asian Test Symposium (ATS), Ehime, Japan, 22–25 November 2021; pp. 67–72. [CrossRef]
32. Gaudesi, M.; Reorda, M.S.; Pomeranz, I. On test program compaction. In Proceedings of the 2015 20th IEEE European Test Symposium (ETS), Cluj-Napoca, Romania, 25–29 May 2015; pp. 1–6. [CrossRef]
33. Cantoro, R.; Sanchez, E.; Reorda, M.S.; Squillero, G.; Valea, E. On the optimization of SBST test program compaction. In Proceedings of the 2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Cambridge, UK, 23–25 October 2017; pp. 1–4. [CrossRef]
34. Zhang, Y.; Li, H.; Li, X. Automatic Test Program Generation Using Executing-Trace-Based Constraint Extraction for Embedded Processors. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2013**, *21*, 1220–1233. [CrossRef]
35. Riefert, A.; Cantoro, R.; Sauer, M.; Reorda, M.S.; Becker, B. Effective generation and evaluation of diagnostic SBST programs. In Proceedings of the 2016 IEEE 34th VLSI Test Symposium (VTS), Las Vegas, NV, USA, 25–27 April 2016; pp. 1–6. [CrossRef]
36. Apostolakis, A.; Psarakis, M.; Gizopoulos, D.; Paschalis, A. A Functional Self-Test Approach for Peripheral Cores in Processor-Based SoCs. In Proceedings of the 13th IEEE International On-Line Testing Symposium (IOLTS 2007), Heraklion, Greece, 8–11 July 2007; pp. 271–276. [CrossRef]