In-field Functional Test of CAN Bus Controllers

In-field Functional Test of CAN Bus Controllers

Riccardo Cantoro, Sandro Sartoni, Matteo Sonza Reorda
Department of Computer and Control Engineering
Politecnico di Torino
Torino, Italy

***Abstract*—The Controller Area Network (CAN) bus is a serial bus protocol widely used in the automotive domain to allow communication between different Electronic Control Units in the car. Being often part of safety-critical systems, the hardware implementing the CAN network must be constantly tested along the system lifetime, even during the operational phase. CAN controllers are relatively complex modules in charge of managing the sending and the receiving of packages through the CAN bus and defects affecting them can easily compromise the whole CAN network. In this work, the CAN controller is tested by test programs to be executed by the CPU connected to the device under test and by another unit connected to the same CAN bus. A fault grading with respect to structural permanent faults of a functional test based on the execution of a software test library for the CAN bus is presented for the first time. Results show how the approach can cover more than 90% of stuck-at faults on an open-source implementation of the standard, which is significantly more than what a usual functional test based on some sample application can achieve.**

***Index Terms*—software-based self-test, software test library, on-line test, automotive electronics, safety**

I. Introduction

Since its introduction in 1986, the *Controller Area Network* (CAN) bus [1] has been widely used in many domains, especially in automotive applications, whenever a robust connection is required to work in a relatively harsh environment. A major incentive to its usage is currently the availability of a huge set of tools supporting it, as well as the integration of standard CAN controllers in many microcontroller units (MCUs). Due to its widespread adoption, CAN controller IP cores are also frequently present in automotive System on Chip (SoC) devices.

Since both MCUs and SoCs including CAN controller IP cores may happen to be used in safety-critical applications, it is important to devise solutions allowing to test them both at the end of the manufacturing process, and during the operational phase (*in-field test*). Standards such as ISO 26262 force the adoption of such solutions and require the quantitative evaluation of the Fault Coverage (FC) that can be achieved in this way, e.g., resorting to the well-known stuck-at fault model, defining challenging targets to be achieved in order to match the specified reliability level [2].

When considering the test of the CPU cores, solutions based either on Design for Testability (DfT) or on Self-Test Libraries (STLs) can be adopted. In the latter case, the device manufacturer provides the user with some test programs, to be integrated into the application code. By activating them during the operational phase (e.g., at the power-on, or during the application idle times), the user can check whether the CPU core is still correctly working, or it is affected by some defect. When developing such STLs, the device manufacturer can precisely assess their effectiveness in terms of achieved structural Fault Coverage (e.g., in terms of the stuck-at fault model). The assessment is typically based on Fault Simulation experiments relying on a new generation of tools, denoted as Functional Fault Simulators. Several semiconductor and IP companies are currently offering STLs to their customers [3]–[9].

Since MCUs and SoCs are also composed of other IP cores besides the CPU ones, such as memory and peripheral ones, the same in-field test approach can be extended to them, as well [10], [11]. In the case of peripheral cores, the idea is to execute a program on the CPU, in charge first of configuring them, and then of forcing them to execute some specific transmission operations. 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 the test relies on another peripheral core of the same type, possibly hosted on another device connected to the same network. During the test, the CPU on the second device will also execute a suitable piece of code, performing symmetrical operations. This scheme has been investigated and successfully assessed in previous works, such as [12].

When moving to the test of a CAN Controller, a similar approach can be adopted, although with some major differences. The first one lies in its size and complexity. Hence, the test is likely to be more complex and longer in terms of execution. A second one is related to the fact that when in-field test must be performed in order to achieve a given reliability or safety level, only faults which can produce a critical failure in the adopted configuration should be considered. Faults that cannot produce any misbehavior are called *Safe Faults* or *Functionally Untestable Faults* (FUFs) [13]. As an example, if some Design for Testability structures have been introduced to support end-of-manufacturing test, most of the faults affecting them are not able to influence the device behavior during the operational phase and can be labeled as FUFs. Previous works on the subject proved that the amount of FUFs may be relatively large, sometimes accounting for more than 20% of the total number of faults. Unfortunately, there are no mature techniques to identify all the FUFs in an automatic

and scalable manner, yet. This fact increases the cost for developing the STLs and makes the computation of the FC they achieve more complex. Finally, given the complexity of the transmission and arbitration protocol it supports, fully testing a CAN Controller may require special techniques which are not required for the test of other peripheral modules.

In this work we selected an open-source model of a CAN Controller, created a sample SoC combining it with an OpenRISC CPU core [14], and developed an STL for the CAN Controller under the assumption that another CAN Controller exists on a second unit in the network and we can run a coordinated test involving the two controllers. The developed STL is composed of deterministic patterns which can be reasonably applied to different CAN Controllers with adjustments but not to other communication peripherals, since the communication protocol may change significantly. Moreover, experimental results gathered on the sample SoC are reported, showing the effectiveness and limitations of the proposed solution. To the best of our knowledge, this is the first work describing a functional solution for the test of a CAN Controller, and the first to report quantitative experimental results about the achievable Fault Coverage. We also compared the achieved FC with the one of a "normal" application using the CAN Controller, showing that the former is significantly higher than the latter. As this is a first step towards an on-line test solution, the assumption is not to rely on any DfT approach (e.g., Logic BIST). We are currently investigating on various in-field scenarios in which some configurations are not allowed, thus leading to untestable logic.

The paper is organized as follows. Section II illustrates a description of CAN controllers, including the version used as the case study of this work. Section III describes in details the proposed approach. In Section IV we present the case study, while in Section V we show experimental results. Finally, Section VI concludes the paper and presents future works.

II. Background

A. CAN Controller

The *CAN BUS protocol*, released in 1986, is a serial communication standard designed with the specific aim of making it robust with respect to noisy environments; it was originally developed for the automotive industry.

The protocol consists of a multi-master bus; there is no need for an arbiter since there can be no problems either from an electrical and logical point of view as this is a *0 dominant bus* and *open drain* technology is employed. In case of simultaneous transmission, the node that is sending a lower priority message (i.e., a collision arises due to the transmission of a logic 1 while another node is transmitting a logic 0) is able to recognize such discrepancy and suspend the transmission.

The CAN standard supports four different frames (i.e., types of message), that are:

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.

As for Data Frame, messages are divided into different fields like 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 which elaborates 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 a register-based interface in which any information can be found.

The actual implementation depends on the producer, here are some of them: SJA1000 by NXP, bxCAN by STMicroelectronics, TI TMS230, and Infineon MultiCan. The following subsection illustrates the case study used in this work.

B. The SJA1000 Implementation

The SJA1000 [15] is a stand-alone controller for the CAN developed by Philips Semiconductors (now NXP Semiconductors) in early 2000s.

It is the successor of the PCA82C200 CAN controller (*BasiCAN*) from Philips Semiconductors. Additionally, a new mode of operation is implemented (*PeliCAN*) which supports the CAN 2.0B protocol specification with several new features.

The peripheral can be described by means of few blocks:

1) *Interface Management Logic*: this block contains a set of registers that implements the peripheral interface plus the logic necessary to interpret the received commands and drive the whole Controller.
2) *Message Buffer*: this block is in charge of providing an interface between the external CPU and other internal modules of the CAN Controller: the Transmit Buffer (TXB) stores the message ready to be sent assembled by the Bit Stream Processor (BSP), the receive buffer stores the received message that comes from the Acceptance Filter. The Receive Buffer (RXB) belongs to the Receive FIFO, that is a FIFO capable of storing up to 64 bytes of messages: from this point of view, the RXB is a window that shifts through the whole FIFO.
3) *Bit Stream Processor (BSP)*: this block consists of a sequencer which controls the data stream between the transmit buffer and the CAN-bus. It also performs the error detection, arbitration, stuffing and error handling on the CAN bus.
4) *Acceptance Filter (ACF)*: this block is in charge of checking whether the message currently on the BUS has

to be received by the peripheral or not. This is assessed by means of an acceptance filter that usually works on the ID, Remote Transfer Request (RTR) bits and DB fields of the message. If the frame passes the filter it goes into the RXFIFO, otherwise it is discarded.

5) *Bit Timing Logic (BTL)*: the BTL block monitors the serial CAN-bus line and handles the bus line-related bit timing. It is synchronized to the bit stream on the CAN-bus on a recessive-to-dominant (i.e., $1 \rightarrow 0$) transition at the beginning of a message (hard synchronization) and re-synchronized on further transitions during the reception of a message (soft synchronization). The BTL also provides programmable time segments to compensate for the propagation delay times and phase shifts and to define the sample point and the number of samples to be taken within a bit time.
6) *Error Management Logic (EML)*: the EML is responsible for the error confinement of the transfer-layer modules. It receives error announcements from the BSP and then informs the BSP and IML about error statistics.

III. Proposed Approach

The approach proposed in this work relies on a distributed software running on at least two devices connected to the same CAN bus. The main reason for this choice relies on the fact that testing the CAN controller of a given node not only requires sending messages on the bus, but also reading messages coming from other devices attached to the same bus. The approach is purely functional and does not add hardware overhead to the system. To the best of our knowledge, this is the first work that deals with the functional test of CAN controllers by means of a distributed software solution.

The test is based on a hardware abstraction layer consisting of ad-hoc drivers that control the bus by means of the low-level operations supported by the CAN Controller. Since nodes can have different architectures, we deal with device-specific drivers. Moreover, drivers do not require additional operations with respect to the standard ones; hence, third party CAN device drivers can be used to implement the approach. Being purely functional and relying on the usage of device drivers, the proposed test solution only stimulates functionally testable faults. This means that precise error conditions on the bus cannot be easily reproduced and this represents a limitation that can be overcome when ad-hoc hardware is available.

In our approach, each node is classified according to three categories:

1) *Neutral nodes*: nodes not involved in the test.
2) *Active nodes*: nodes under test, which are executing a test program and are the main controllers of the bus. Such nodes send messages or receive messages from passive nodes.
3) *Passive nodes*: nodes that are executing a test program to support the test of active nodes. Such nodes ignore or react to messages received from active nodes, and send messages according to the specifications of the test program.

The test program is installed both in the active and passive nodes. Each program thus comes in two fashions: depending on whether a node is active or passive it will transmit first or receive first the messages.

The main test program has been developed with the goal of testing every possible functional configuration and it is divided into sub-programs (or parts), each targeting a specific sub-module or functional feature of the CAN controller. They can be executed together as a whole, grouped in sub-modules or by themselves one after the other: the idea is that, when the CAN bus is idle, the test can occur and depending on the duration of the idle time one could decide on the test length.

In the rest of the Section the reader can find a description of the sub-programs that compose the STL.

A. Bit Rate test

One of the key parameters in the CAN bus is the transmission bit rate. In this sub-program, the active node sends a message using multiple bit rates, without varying the other configuration parameters. In our experiments, we set the CAN controllers to work in BasicCAN mode and we fixed the other parameters, e.g., the ID and the data length (DLC) fields. For each message sent, the active node expects an acknowledgment from the passive node, which changes the transmission bit rate coherently. Once tested the ability of the node to work at various bit rates, the nodes are configured to the fastest possible bit rate for all the remaining phases of the test.

B. Normal Mode test

This sub-program is intended to test the basic transmission/reception of messages while changing the configuration parameters. Clearly, active and passive nodes change parameters coherently during the test.

The other parameters are changed, such as the CAN operating mode (i.e., BasiCAN and PeliCAN), the Frame Format when in PeliCAN mode, the DLC and ID and the enabled interrupts. One or more messages are exchanged in each of the available configurations. In our experiments, data payload for messages have been filled with pseudo-random values. Additionally, we included some deterministic patterns (e.g., 0101..., 1010..., 0011..., 1100...). After each transmission, the results are retrieved by reading the appropriate registers (i.e., data and status registers) and compared with the expected ones (or compacted in a test signature). Finally, configuration registers are read back after each change in the configuration to detect faults in the configuration flip-flops.

C. Self-Test Mode test

In the Self Test Mode configuration, the CAN controller sends messages without the need of an acknowledge by other nodes and uses a loop-back to check their correctness autonomously. The test consists of a transmission and concurrent reception of a certain amount of messages (100 in our experiments), all of this in PeliCAN Mode.

D. Listen Only Mode test

In the Listen Only Mode configuration, the CAN controller is only capable of receiving and, more specifically, it does not generate the acknowledge bit even if the message has been correctly received. The test aims at checking the ability of the node to receive and process a message and consists in the reception and check of a certain amount of messages (100 in our experiments) sent by the passive node.

E. FIFO test

The basic principle of the sub-program for the FIFO consists of filling and then emptying it and working on the overrun generation bit. The overrun occurs when the FIFO is already full and the CAN controller tries to write another message. In order to implement such a principle, the passive node has to send enough messages to fill the FIFO (64 messages in our case study). A further message sent by the passive node produces an overrun, which can be detected by the active node by reading a proper status register (or by means of an interrupt). In a second phase, while the second node is sending messages, the first one keeps reading them, in order to test the remaining logic of the FIFO. In our case study, the FIFO is implemented as a circular buffer, thus we sent a first packet of 64 messages and then we repeated this with 128 messages.

F. Errors test

This sub-program aims at testing the logic devoted to detect some error conditions. Since in a functional test environment not assisted by ad-hoc hardware it is not possible to precisely work on external (i.e., coming from the physical BUS) errors, the program mainly focuses on testing the situation in which the two nodes do not have the same bit rate.

The proposed sub-program configures the active and the passive nodes to different bit rates. Then, the passive node sends a message to the active node, which is incapable of receiving it due to the different bit rates. After a certain amount of trials, the active node disables itself and goes into *BUS Off Mode*. Finally, the active node becomes the transmitter and the passive node the one who is subject to the error.

G. Arbitration test

This program tests the arbitration between nodes. In order to test this condition in a deterministic way, the approach consists of a series of messages being sent one after the other. The two messages should be exactly equal in an initial portion and then differ only by one bit, which triggers the loss of arbitration. Reproducing such deterministic behavior in a functional environment is not straightforward and requires to synchronize the end-to-end communication, e.g., using timers to compute the latency due to physical characteristics of the CAN network

H. Acceptance Filter test

Every message received by the CAN controller is filtered by comparing its ID against some programmable bit-masks. In order to test the comparators in the acceptance filters, deterministic patterns can be used [16]. Alternatively, patterns can be easily derived by launching an Automatic Test Patterns Generation tool on the combinational logic existing inside this hardware block. Each pattern is then transformed into a message sent by the passive node and filtered by the active node accordingly.

In case of no available passive node, the active node will have to rely on the Self-Test mode, only. As a consequence, a new list of untestable faults due to the constraints of the test configuration should be identified.

IV. Case Study

The CAN controller used as the case study is freely available on OpenCores [17] and implements the SJA1000 Controller [15]. The author of the design claims the CAN Controller has been tested in hardware and verified with the Bosch VHDL Reference System [18]. The RT-level design was synthesized with the Synopsys Design Compiler using a 65nm technology library. The final netlist accounts for 12,953 NAND2 equivalent gates, including 1,460 flip-flops and 38,490 stuck-at faults.

For the purposes of this work, we inserted the CAN controller into a SoC also composed of an OpenRISC core OR1200, available on OpenCores [14]. The controller in hand officially supports the internal Wishbone Bus, thus it was attached besides the other available peripherals without the need of any bridging circuitry. Moreover, the internal registers have been memory-mapped to a portion of the available memory space, thus allowing the SoC to access the CAN interface by means of load and store instructions.

Since the CAN controller from OpenCores comes with no drivers, we implemented a CAN driver written in C language. In order to give an idea of the kind of operations required by the test, in the following we report the main functions implemented by the driver:

- $canPeriphInit$: it initializes the CAN peripheral by setting the correct mode between BasiCAN and PeliCAN, it configures the registers *BUSTIMING0/1*, which control the timing behaviour of the peripheral, and *Acceptance Code/Mask*, used to decide which IDs are accepted or not when receiving a message.
- $irqEnable$: it enables/disables the interrupts supported by the CAN peripheral.
- $transmitMsg$: it sets the registers needed to send a message - namely, *TXDATA0* to *TXDATA12* - and then it starts the transmission in polling mode, with a timeout which aborts the transmission if the message was not sent after a certain number of cycles.
- $receiveMsg$: it checks the register *Receive Buffer Status* (i.e., it tells whether there is an available message or not) in polling mode and, if the timeout is reached and there is still no message, it exits. In the other situation, the functions extracts the message and saves it in a data structure, which contains all the fields necessary to the user.

- $selfTxRx$: it is used when in *Self-Test Mode*. It tells the peripheral to send a message receiving it at the same time without the need of an external acknowledge.

In the test-bench used in our experiments, two instances of the OR1200-based SoC are connected through a CAN bus. The whole system was simulated at the RT level, with the exception of the CAN controller of the target SoC which is simulated at the gate level. Since the test-bench implements an end-to-end communication between two devices, the two SoCs own distinct IDs. For simplicity, test programs are loaded after the boot of the system and are configured in active and passive mode accordingly. A simple scheduling approach is implemented, which iterates on all the available sub-programs.

V. Experimental Results

The fault grading has been conducted by means of a commercial fault simulator. Functional patterns have been derived by logic simulation and dumped in a VCD file. The VCD file has then been applied to the top-level of the SoC that embeds the CAN controller in a full sequential fault simulation. We periodically checked the output values produced by the SoC on the bus as observation points.

The details about the fault simulation experiments are reported in Table I. The table reports, for the main sub-modules in the CAN controller, the number of stuck-at faults and the fault coverage. Since many smaller modules and glue logic are not reported in the table, the number of faults of a given module is higher than the sum of the faults in the reported sub-modules. On the top-level module of the CAN controller, we covered 90.22% of faults.

Details about the parts that compose the test program described in Section III are reported in the upper part of Table II. The table reports, for each part, its test application time in clock cycles (column 2), the number of messages exchanged by the nodes (column 3), and the amount of bytes sent by the active node (column 4) and passive node (column 5). The clock signal period adopted in the testbench is 10 ns; as a consequence, by grouping together all the parts, the total amount of time required by the whole test program to run is about 555 ms. The remaining columns report the fault coverage on the main sub-modules (the last column refers to the full fault list) reached by running the parts separately. Finally, the last row of the upper part reports the cumulative fault coverage values.

The results highlight how for some modules the contribution to the cumulative fault coverage is given by a specific part (e.g., registers are mainly tested by the Normal Mode part, as well as the Acceptance Filter is mainly tested by the related part), while for some others, the contribution is spread among the parts (i.e., the BTL and BSP modules, mainly). The reader can also notice that the FIFO is well tested, as a side effect, by parts written for other purposes, thus some tests could be potentially shortened.

TABLE I: Faults Report of the CAN Controller

Instance Name	#Stuck-at Faults	Fault Coverage %
i_can_registers	$5,352$	84.91
i_can_btl	$1,472$	83.24
i_can_bsp	$31,236$	91.43
i_can_acf	$1,418$	85.40
i_can_fifo	$17,382$	96.26
TOTAL	$38,490$	90.22

To gain some insight on the validity of the proposed method, we compared it to alternative test programs, mimicking what a "normal" application using the CAN Controller does, and reported the results in the lower part of Table II. Such test programs consist in random messages sent and received in Normal Mode. In the first two cases in the table, we send variable size messages while keeping all the configuration parameters fixed. In the last two cases, we also vary the configuration. The table also reports the cumulative results of such alternative test programs. Additionally, since they only work in Normal Mode, in the last row we added the contribution of the parts developed for Self-Test Mode (STM) and Listen Only Mode (LOM).

The results show that a significant increase in the amount of messages is not an effective strategy for modules such as Registers, BTL, and ACF, while it works well for BSP and FIFO. On the contrary, Registers and BTL are more susceptible to the configuration changes. The ACF resulted to be random resistant. The proposed approach outperforms the cumulative results of 71.08% reached by the alternative solutions by nearly 20 percentile points (16 if we include the Self-Test Mode and Listen Only Mode parts).

A careful analysis of the faults not detected by the proposed approach has shown that part of them (e.g., inside the ACF or the FIFO) can be covered by ad-hoc messages. However, a significant amount of faults were hard to test due to the hardware configuration, such as those in the logic related to error and arbitration lost conditions (e.g., in the BSP).

Errors such as those due to electromagnetic interference or physical problems on the bus cannot be emulated by means of two nodes connected with an ideal wire and thus the test of the related logic would require an external module capable of sensing the line and pulling it down at a given time. If this approach was adopted in the system (with the related complexity and additional hardware and design cost), the error conditions and the logic in charge of detecting errors could be tested more thoroughly.

Concerning arbitration lost, since the CAN protocol does not provide an arbiter and the CAN Bus is 0 dominant, arbitration is achieved by means of checking the Rx wire while sending the message: for each bit, if the bit sent is different than the one on the bus it means that a higher priority message is being sent at the same time and so the node with the lowest priority aborts the transmission. The arbitration lost could occur in any part of the message. The main problem here is trying to synchronize the transmission of the two peripherals and dedicated hardware should be implemented to precisely break messages sent by the active node.

Finally, there are limitations due to the sampling precision,

TABLE II: Comparison between the proposed sub-programs and alternative solutions

	Test program	Duration [clock cycles]	#Messages	Bits sent	Bits received	Registers FC%	BTL FC%	BSP FC%	ACF FC%	FIFO FC%	*TOTAL* FC%
Proposed approach sub-programs	Bitrate part	1,577,790	8	864	864	39.16	68.84	42.59	3.74	39.60	*43.66*
	Normal Mode part	248,652	8	1,144	1,144	70.98	51.03	38.40	17.49	24.09	*44.03*
	Self-Test Mode part	10,237,885	100	3,536	3,536	46.29	67.95	28.93	10.16	10.35	*33.48*
	Listen Only Mode part	10,237,885	100	0	3,536	26.47	67.60	60.84	13.82	77.30	*56.63*
	FIFO part	3,157,031	328	0	28,000	25.37	51.64	60.46	6.77	81.47	*55.58*
	Errors part	410,452	1	72	72	30.93	69.45	14.93	0.42	0.35	*19.96*
	Arbitration part	815,000	30	2,160	2,160	40.00	52.12	17.44	3.95	0.77	*22.56*
	Acceptance Filter part	28,856,547	592	44,176	44,176	48.18	51.71	69.45	81.95	80.26	*66.05*
	ALL	55,541,242	1,167	51,952	83,488	84.91	83.24	91.43	85.40	96.26	*90.22*
Alternative solutions	20 random messages	442,063	20	640	640	37.83	51.58	47.13	3.10	48.56	*46.48*
	200 random messages	4,399,865	200	6,400	6,400	37.17	51.58	61.95	3.10	75.11	*58.42*
	20 random configurations	2,841,090	20	312	312	44.60	70.96	42.95	4.58	37.90	*44.80*
	200 random configurations	20,046,684	200	3,536	3,536	45.59	74.38	65.26	4.58	77.53	*63.17*
	ALL	27,729,702	440	10,888	10,888	52.63	74.38	73.80	4.58	86.66	*71.08*
	ALL + STM and LOM parts	48,205,472	640	14,424	17,960	62.32	77.05	75.99	17.91	87.72	*74.31*

which impact on the coverage of the BTL. Even thought it is possible to act on the programmable time segments in the BTL, some values cannot be actually used in functional mode.

To summarize, most of the faults left untested by the proposed method cannot be physically detected resorting to a test based on simply connecting two CAN Controllers and forcing them to communicate.

VI. Conclusions

We presented the first work that specifically targets the structural test of CAN controllers by means of test programs installed on two devices connected to the same CAN bus. Using this approach, one can test at speed and in the field the CAN Controller without adding special DfT features (hence, without area overhead). Moreover, the proposed test approach, being functional, avoids any overtesting. The paper gives guidelines about the messages to transmit on the CAN bus to cover the functional blocks in the controller. Experiments on a system composed of two open-source SoCs connected to the same CAN bus show that 90.22% of the stuck-at faults in the CAN controller can be covered by the proposed approach. We believe that there is a significant fraction of the untested faults which are functionally untestable, and are working to identify them in a provable manner. Given its flexibility and high reusability, the proposed software-based approach is very well suited to be used for in-field test of CAN controller modules embedded in safety-critical systems. Future activities are planned to cover hard-to-test faults by exploiting more complex configurations (e.g., using more than two nodes). Moreover, timing faults will be targeted by future works, as well as faults affecting the CAN bus due to electrical problems. Finally, the handling and the scheduling of the end-to-end test programs in production will be studied. One further improvement could be done in terms of application time, since this was not the main concern of the work.

Acknowledgments

The work has been supported by the Center for Automotive Research and Sustainable mobility@PoliTO (CARS).

References

[1] Robert Bosch GmbH., "CAN Specification, Version 2.0," 1991. [Online]. Available: http://www.semiconductors.bosch.de/pdf/can2spec.pdf

[2] Ross, *Functional Safety for Road Vehicles: New Challenges and Solutions for E-mobility and Automated Driving*. Springer International Publishing, 2016.

[3] Hitex, "Microcontroller self-test libraries." [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/

[4] STMicroelectronics, "Guidelines for obtaining IEC 60335 Class B certification for any STM32 application," Mar 2016. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/application_note/02/1a/91/78/e4/15/4d/35/CD00290100.pdf/files/CD00290100.pdf/jcr:content/translations/en.CD00290100.pdf

[5] Cypress Semiconductor, "FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library." [Online]. Available: https://www.cypress.com/file/249196/download

[6] Renesas Electronics, "SSP Supplemental Add-Ons." [Online]. Available: https://www.renesas.com/en-eu/products/synergy/software/add-ons.html

[7] Microchip Technology Inc., "16-bit CPU Self-Test Library User's Guide," 2012. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[8] ARM, "Enabling Our Partnership to Bring Safer Solutions to the Market Faster." [Online]. Available: https://developer.arm.com/technologies/functional-safety

[9] NXP Semiconductors, "S32 SDK for S32K1 microcontrollers." [Online]. Available: https://www.nxp.com/support/developer-resources/run-time-software/s32-sdk/s32-sdk-for-s32k1-microcontrollers:S32SDK-ARMK1

[10] 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, March 2009.

[11] van de Goor *et al.*, "Memory testing with a risc microcontroller," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, March 2010, pp. 214–219.

[12] Bolzani *et al.*, "An automated methodology for cogeneration of test blocks for peripheral cores," in *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, July 2007, pp. 265–270.

[13] Cantoro *et al.*, "An analysis of test solutions for COTS-based systems in space applications," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2018, pp. 59–64.

[14] "OpenRISC Project Overview." [Online]. Available: https://openrisc.io/

[15] Philips Semiconductors, "Stand-alone CAN controller," 2000. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/SJA1000.pdf

[16] Grigoryan *et al.*, "Generic bist architecture for testing of content addressable memories," in *2011 IEEE 17th International On-Line Testing Symposium*, July 2011, pp. 86–91.

[17] "CAN Protocol Controller Overview." [Online]. Available: https://opencores.org/projects/can

[18] GmbH., "Automotive Electronics VHDL Reference CAN." [Online]. Available: http://www.bosch-semiconductors.com/media/ip_modules/pdf_2/vhdl_reference_can/bosch_ip_info_vhdl_reference_can.pdf