Effective techniques for automatically improving the transition delay fault coverage of Self-Test Libraries

Riccardo Cantoro, Francesco Garau, Patrick Girard, Nima Kolahimahmoudi, Sandro Sartoni, Matteo Sonza Reorda, Arnaud Virazel

Effective techniques for automatically improving the transition delay fault coverage of Self-Test Libraries

Riccardo Cantoro*, Francesco Garau*, Patrick Girard†, Nima Kolahimahmoudi*,
Sandro Sartoni*, Matteo Sonza Reorda* and Arnaud Virazel†

*Department of Computer and Control Engineering
Politecnico di Torino
Turin, Italy

†LIRMM
University of Montpellier / CNRS
Montpellier, France

***Abstract*—In-field test of integrated circuits using Self-Test Libraries (STLs) is a widely used technique specifically suited to guarantee the processor's correct behavior during the operative lifetime, as mandated by functional safety standards such as ISO26262. Developing STLs for stuck-at faults requires significant manual efforts from test engineers, and targeting delay faults is even more challenging. In order to support this process, in this paper we propose a method to automate the creation of STLs targeting delay faults starting from existing STLs targeting stuck-at faults. The method is based first on identifying excited but not-observed transition delay faults and then adding suitable instructions able to observe (and hence detect) them. Experimental results on a RISC-V processor show that the method can systematically detect a significant percentage of the target faults with reasonable computational effort and test code size increase.**

***Index Terms*—software-based self-test, software test libraries, in-field test, safety, functional test, delay faults**

I. Introduction

New advanced semiconductor technologies are increasingly adopted in emerging applications, thanks to their enhanced working frequencies and computational capabilities. Such technologies, however, are extremely complex and sophisticated, leading to more frequent physical defects and reduced operative lifetime. Testing integrated circuits (ICs), hence, is of paramount importance. Most of these defects are tested by targeting not only static, but also dynamic defects, often modeled as delay faults, i.e., faults that affect the timing behavior of the device under test (DUT), such as transition delay faults (TDFs) or path delay faults (PDFs). Testing integrated circuits can be done using two different approaches. The most common one relies on the adoption of Design-for-Testability (DfT) solutions, which usually require the usage of additional hardware modules such as Logic BIST or scan chains. Such modules are integrated within the DUT and are employed to apply test vectors and monitor the circuit's response to the aforementioned vectors. Although based on mature technology and supported by most EDA tools, such solutions impose non-negligible timing and area overheads that could degrade performances. Moreover, functionally untestable faults [1] (FUFs), i.e., faults whose effects can never be observed within functional scenarios, will possibly be detected, leading to a phenomenon known as *overtesting*, which leads to a yield loss. These issues can be overcome by adopting another testing solution, namely functional testing. In the form of Software-Based Self-Test (SBST), functional testing [2], [3] is based on the execution of a set of Self-Test Libraries (STLs) by the DUT. The results produced by the test programs are compacted into a signature that is compared against the golden circuit's one to look for the presence of structural faults. This approach has been proved effective both when processor cores [4]–[11] and peripherals [12]–[15] are tested, and several companies provide STLs for their products [16]–[19]. SBST is a desirable solution for *in-field testing*, i.e., when the device's reliability and safety has to be guaranteed throughout the operative lifetime. SBST is reliable, cheap, and flexible — STLs can be developed such that they fit the idle slots of the application run by the DUT, hence avoiding any service interruption. Thanks to these properties, SBST can be successfully used whenever compliance to standards such as the *ISO26262 standard* for automotive systems is required.

However, developing STLs from scratch is not a trivial task, more so when targeting delay faults on complex devices. For test programs to achieve high fault coverage figures, they must be able to excite as many faults as possible from the whole DUT and make their effects observable at primary outputs (POs) by using instructions from the system's instruction set architecture, only [20]. Achieving this requires a non-negligible amount of manual effort by the test engineer. Moreover, when fault reports are available, understanding why certain faults are not detected (possibly isolating the contribution of FUFs) is not always easy. The work in [21] moves the first step in classifying not-observed transition delay faults, giving some insights on where these fault effects propagated and stopped and defining some upper boundaries on how much the final TDF coverage can be increased.

In this paper, we propose an automatic and systematic methodology to increase the TDF coverage of STLs, by detecting faults identified in [21] on complex pipelined processor cores, starting from a set of test programs devised for stuck-at faults (SAFs). We choose the transition delay fault model over the path delay one because TDFs are much better supported

Authors are listed in alphabetical order.

by both standards and EDA tools than PDFs. The main contributions of this work are:

- a set of techniques able to identify which instructions are capable of detecting not-observed transition delay faults;
- a test flow able to automatically add the previously identified instructions into the right place within existing STLs to improve the final fault coverage;
- data on how much overhead is added to the original STL after its enhancement.

This approach is validated on a RISC-V core, using available commercial tools and a set of pre-existing STLs targeting SAFs. The reported results show that it is possible to detect most of the aforementioned transition delay faults (increasing the TDF coverage by up to 15%) with a reasonable computational effort and test time increase.

The article is organized as follows: in Section II a background on the transition delay fault model and related works is outlined, while in Section III we describe the proposed approach. In Section IV we present the experimental results and, finally, in Section V we draw the conclusions.

II. Background

A. Transition delay faults

Transition delay faults are a class of faults that causes failures in the timing behavior of a circuit. They are modeled so that large delays are concentrated at a logical node, differently from path delays faults that are modeled by means of delays distributed on a path. Their effects are noticeable when the circuit works at nominal speed, usually affecting the propagation delay of signals. TDFs come in two different types, namely slow-to-rise (STR) — i.e., faults that affect rising ($0 \rightarrow 1$) transitions — and slow-to-fall (STF) — i.e., faults that affect falling ($1 \rightarrow 0$) transitions.

Testing TDFs, hence, requires applying rising and falling transitions on the node to be tested, and propagating the transition to an observable point of the circuit under test. Consequently, tests are comprised of two pairs of vectors: one for initializing the signal, the other for generating the intended transition and propagating its effects. Current EDA tools well support the transition delay fault model when resorting to DfT approaches (that usually involve the usage of scan chain-based protocols like Launch-on-Shift and Launch-on-Capture). Functional solutions, however, are also a particularly effective approach, especially when considering that test vectors for testing delay faults must be at speed, a feature that is easily achieved when executing STLs. In this case, test vector pairs are provided by the instructions within the test program, which must be able to excite TDFs and propagate their effects towards POs. Unfortunately, EDA tools do not currently provide effective solutions for automatic test program generation, which is still an open issue in the testing community.

B. Related Works

The development of test programs for delay faults through an SBST approach has been faced by some works describing methodologies to do so [6], [8], [10]. Regarding TDF specifically, [21] introduces a study on transition delay faults of modern pipelined CPUs that have not been observed throughout the execution of STLs targeting SAFs, focusing on where their effects propagated and stopped inside the DUT. The article introduces two fault groups: *User Accessible Register* faults (UARs), whose effects reached registers that can be directly observed through instructions from the CPU's instruction set architecture, e.g., the register file, and *Hidden Register* (HRs) faults, whose effects reached registers non directly controllable, e.g., pipeline registers. The work in [21] provides some useful insights on what faults to target to improve the final fault coverage, and provides an upper bound on how much the final transition delay fault coverage can be improved. Given three test programs, namely STL1, STL2 and STL3, [21] shows that it is possible to increase their fault coverage by, relatively, 9.15, 17.85 and 8.96 percentile units. This work, however does not provide strategies to detect them, which is the goal of this paper. The work [12] describes STL development strategies for peripherals embedded in modern System on Chips, achieving significant fault coverage figures. Works [4], [5], on the other hand, focus on the development of test programs for processor cores. [4] describes a methodology to test delay faults on computational blocks within superscalar processors, while [5] aims at testing RISC-like CPUs by dividing them into modules under test and devising test strategies for each of these modules without the need of knowing implementation details. Finally, [22] presents a reinforcement learning-based test program generation technique for TDFs validated on a MIPS32 core. Although effective, showing that it is possible to thoroughly test delay faults through functional means, all these works require the generation of test programs starting from scratch, a task that requires non-negligible time and effort from test engineers. [22], moreover, requires the usage of a reinforcement learning algorithm, which might not be particularly effective when tackling high-complexity cores.

Articles like [23]–[25] focus on the improvement of available programs to obtain high fault coverage figures. [23] describes how to derive test patterns intended for online testing starting from programs originally intended for verification purposes, significantly increasing the final coverage of stuck-at faults on a RISCV core. Works [24], [25], on the other hand, present a tool based on High-Level Decision Diagrams (HLDDs) for modeling microprocessors and faults, used in conjunction with previously prepared code templates to generate the final self-test program targeting stuck-at faults. These works show that methodologies for improving test programs can be successfully devised. Nonetheless, they are developed bearing the classical stuck-at fault model in mind.

The goal of this paper is to propose some techniques allowing to automate the transformation of existing STLs targeting SAFs so that the resulting TDF coverage is improved.

III. Proposed Approach

This work mainly stems from an empirical observation: STLs targeting stuck-at faults often fail in achieving a high

TDF coverage because of their inability to propagate TDF effects up to some observable point. By improving the ability of an STL to propagate TDF effects we could significantly increase its TDF coverage.

Detecting transition delay faults that were excited but not observed by STLs requires some considerations on where their effects propagated and stopped within the DUT. The reason for doing so lies in the fact that, in order to detect the aforementioned faults, we need to propagate their effects towards observable points (e.g., memory locations) and strategies for such task may vary based on the functional sub-module from which the propagation occurs. Given this premise, in this work we start from the two main categories of internal observation points introduced in [21], i.e., User Accessible Registers (UARs) and Hidden Registers (HRs), and define some internal observation points to be used during simulation to further refine the topological analysis about fault effects. Inserting observation points is done by exploiting capabilities offered by available commercial fault simulation tools. Such task does not require the modification of the DUT's hardware as they are simply labels that the tool attaches to the netlist. To each label a fault status is associated, and the hierarchy of the statuses can be customized to make sure that locations closer to the primary outputs of the architecture are prioritized. As a further step, this paper describes how to introduce suitable instructions in the STL code, so that for each group of faults, a significant percentage of them is made observable, and hence detected.

In the following, we discuss strategies for the aforementioned two main fault categories.

A. *User Accessible Register faults*

User Accessible Register faults are faults whose effects have been propagated from the original fault site to registers that can be directly observed through instructions from the instruction set architecture. Being able to directly access these registers' content through instructions helps the test engineer to make faults effects observable at the primary outputs.

In order to detect such faults, first we must analyze the fault data base produced at the end of the fault simulation where internal observation points have been added. Such data base stores information on which faults have been observed at any given internal observation point at some specific time instant. This allows us to obtain topological information, i.e., what register we have to work on, as well as chronological information, i.e., what portion of the test program should be improved. The latter is possible since we are able to associate instructions being executed by the CPU to the simulation time reported in the dictionary.

When analyzing the time at which fault effects reached user accessible registers, it is crucial to keep in mind that, in modern in-order pipelined processor cores, there are instructions that take more than one clock cycle to go through the CPU execution stage, e.g., division operations. These instructions — from hereinafter referred to as *multi-cycle instructions*, as opposed to *single-cycle instructions* that only take one cycle in

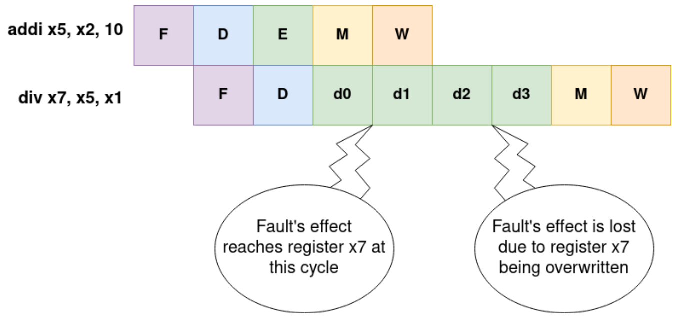
addi x5, x2, 10
F
D
E
M
W
div x7, x5, x1
F
D
d0
d1
d2
d3
M
W
Fault's effect reaches register x7 at this cycle
Fault's effect is lost due to register x7 being overwritten

Fig. 1. Multi-cycle instructions and fault effects propagation

the execution stage — should be carefully taken into account, as the fault effect might be overwritten during the required execution cycles. For instance, given an instruction that takes 4 clock cycles to go through the execution stage, a situation similar to the one reported in Fig. 1 might occur, where the fault effect is propagated at an inner cycle only to be overwritten later, losing the possibility of observing the faulty value. For this reason, we define two strategies depending on whether we deal with single-cycle or multi-cycle instructions:

- *single-cycle instructions*: detecting these faults is quite easy, as it is sufficient to perform a **store** operation on the register affected by the faulty value after the time at which said effect reached the register, and before the register is overwritten by another operation;
- *multi-cycle instructions*: if the fault effect is still present at the last execution cycle the same strategy adopted for *single-cycle instructions* is used, else we modify the operands of the multi-cycle instructions — either arithmetical or logical operations — to ensure that the faulty value reaches the register towards the end of the execution stage, so that it can be observed through a store instruction. As it is demonstrated by the experimental results we gathered, identifying suitable operands for this purpose is a feasible task, which can often performed following a try-and-error approach.

B. *Hidden Register faults*

All those faults whose effects reach registers that cannot directly observed through instructions fall within the Hidden Register faults group. These registers are deeply embedded inside the processor core, either belonging to pipeline registers or inner sub-modules, which makes particularly hard to propagate values from those locations to either primary outputs or user accessible registers (in this case the techniques described in the previous sub-section can then be adopted).

The strategy we propose to detect these faults starts off similarly to Section III-A, that is, by analyzing the fault data base to extract information on where faults propagate and stop and at what time instant, i.e., in what portion of the STL, such events occur. Given the nature of hidden registers, however, additional analysis are needed to understand how to detect these faults.

In order to do so, we observe that the process of exciting a transition delay fault and observing its effects in a pipelined

Algorithm 1: HR faults detection algorithm

```
input : A list $L$ of triplets ($F_i$, $H_i$, $T_i$) where
          $F_i$ is the transition delay fault to be tested
          $H_i$ is the HR bit reached by the fault's effect
          $T_i$ is the time at which the fault effect reached $H_i$
        An STL $S$ that has been developed for SAFs
output: A set of instructions to propagate transition delay
        fault effects to primary outputs
foreach ($F_i$, $H_i$, $T_i$) in $L$ do
    if $S$ detects $H_i$'s stuck-at-1 and/or stuck-at-0 faults then
        get the time $T_s$ at which the stuck-at fault on $H_i$ is
          detected;
        extract a block $B$ with the last $N$ instructions before
          $T_s$ from $S$;
        check whether $B$ does not contain jump instructions;
        if $F_i$ has been detected by $B$ then
            add $B$ to the original program;
        end
    end
end
```

CPU can be decoupled into two sub-processes. First, a specific pair of test vectors must be applied to generate the required transition and propagate it towards an endpoint, may that be a primary output — in which case the fault is marked as detected — or a register within the processor core. Secondly, if the fault's effect reached a register, methodologies to propagate such effect to primary outputs are employed to detect the fault. While the first step obviously depends on the fault model and the transition that we want to generate, the second step does not depend as much on the fault to be excited, and is just a problem of propagating a value from one point to another. The aim of this work is to define an automatic way to easily increase the transition delay fault coverage for STLs that were previously devised for stuck-at faults. Given this group of faults, hence, we define the algorithm summarized in Algorithm 1.

The basic idea behind this algorithm is that the available STLs may already be able to test stuck-at-0 and stuck-at-1 faults located in pipeline registers. The code that serves that purpose, however, can also be used to propagate values from said locations to primary outputs. This makes for an effortless way to detect transition delay faults, as we just need to find the appropriate chunk of code and put it right next to the one that excites the transition delay fault and propagate its effect up to the relative pipeline register. This operation, however, should not disrupt the overall flow of the original test program: for this reason, jump instructions in the code to be added should be avoided. In the rare case when the TDF propagated to a bit in a pipeline register, whose corresponding SAFs were still not detected by the existing STL, methods such as [26] can be used to generate the required chunk of instructions (improving the SAF coverage as well).

A final point that applies to both User Accessible Register and Hidden Register faults is that, given the right premises, a set of instructions added to the original test program may be capable of detecting more than one TDF at the same time. This is only possible for all those transition delay faults whose effects propagate to the same register at the same time. Thanks to this feature, it is possible to achieve better fault coverages with a smaller test program with respect to having a set of instructions for each fault to be tested.

IV. Experimental Results

A. Case study

The methodology introduced in this paper has been validated on PULPino [27], a 32-bit RISC-V-based SoC platform developed by ETH Zurich and Università di Bologna. The DUT has been synthesized using the 45nm Silvaco Open Cell library [28] and accounts for 51,001 NAND2-equivalent gates, 159,326 stuck-at faults (SAF) and transition delay faults (TDF), and 1,207 flip-flops belonging to hidden registers.

As for the test programs, we adopted three different STLs that were originally intended to test SAFs on the PULPino core, namely STL1, STL2, and STL3. In order to ensure a diverse and realistic testbench, the three test programs we selected have been developed following different implementation strategies, by different test engineers. A summary of the most important characteristics of the adopted STLs, namely the execution time (expressed in the total amount of clock cycles), memory size (in kB), and SAF coverage, is reported in Table I.

TABLE I
STLs general information

Test Program	*#Clock cycles*	*Memory size [kB]*	*SAF coverage %*
STL1	17,308	27.32	81.42
STL2	31,158	27.86	81.86
STL3	80,455	16.68	82.18

It is noted that the reported amount of clock cycles is obtained by executing all STLs completely; depending on the situation, the test engineer can then decide to split them into sub-modules that can be launched separately, each requiring a fraction of the overall time with the same final fault coverage. Fault simulations have been carried out using Synopsys Z01X, a commercial tool devised specifically for functional safety. Experiments have been conducted by means of Python scripts, with the goals of collecting information from fault dictionaries, improving test programs according to the methodologies described in Section III, and launching the actual fault simulations. As a result, the full flow of STL improvement and fault simulation for transition delay faults took no longer than 4 days on an Intel Xeon CPU E5-2680 v3 server with a clock frequency up to 3.3GHz.

B. Achieved results

In this subsection we describe the achieved results in details. Table III and Table IV show summary data on the user accessible register (UAR) and hidden register (HR) faults that were detected as a result of the proposed methodology.

Starting with Table III, it is possible to see that our approach is greatly effective as it is capable of detecting almost

TABLE II
SUB-MODULES ANALYSIS FOR THE ADOPTED STLS

Test Program		Hidden Register faults				User Accessible Register faults	
		Fetch Stage	Decode Stage	Execute Stage	Memory Stage	GPRs	SPRs
STL1	Detected faults	23	587	32	1	4, 359	2, 219
	Total faults	1, 109	5, 109	388	135	4, 359	2, 232
	Added Instructions	45	550	25	5	1, 107	478
STL2	Detected faults	52	120	5	6	23, 814	98
	Total faults	1, 311	1, 976	221	91	23, 814	108
	Added Instructions	80	115	20	15	1, 022	20
STL3	Detected faults	13	595	0	0	2, 853	11
	Total faults	1, 028	2, 463	351	113	2, 853	47
	Added Instructions	35	195	0	0	877	6

every fault, with the worst case scenario being STL3 with a 98.76% of UAR faults being detected. Given a total amount of 159, 326 transition delay faults, through our methodology we can increase the final fault coverage by 4.13% for STL1, 15.01% for STL2, and 1.80% for STL3, respectively.

TABLE III
ANALYSIS ON DETECTED UAR FAULTS

	STL1	STL2	STL3
Detected UARs	6, 578	23, 912	2, 864
Total UARs	6, 591	23, 922	2, 900
%Detected UARs	99.80	99.96	98.76
Code size [kB]	6.34	4.17	3.53

This improvement comes with an increase of the final code size, which amounts to an additional 22.21% for STL1, 14.97% for STL2, and 21.16% for STL3. This proves that our strategy is able to systematically test not-observed transition delay faults whose effects reached user accessible registers.

Moving on to Table IV, it is possible to see that the results we achieved thanks to our methodology are quite dependent on the considered STL.

TABLE IV
ANALYSIS ON DETECTED HR FAULTS

	STL1	STL2	STL3
Detected HRs	643	183	608
Total HRs	6, 741	3, 599	3, 955
%Detected HRs	9.54	5.08	15.37
Code size [kB]	2.60	0.92	0.92

For this group of faults, the worst case scenario is represented by STL2, for which 5.08% HR faults can be detected, while the best case scenario is represented by STL3, with a total of 15.37% faults detected. Although the results are not as high as for UARs, it is still worth mentioning that our methodology allows to automatically detect these faults, thus not requiring any manual effort from the test engineer. For this latter group, the increase in the code size is rather small, amounting to an additional 9.52% for STL1, 3.30% for STL2, and 5.52% for STL3, respectively. It is also worth mentioning that some of the undetected faults may belong to the group of FUFs.

Table II describes the information regarding sub-modules of the tested processor core in details, reporting the contributions in terms of detected faults, total faults and added instructions for each sub-module and STL. All the pipeline stages columns belong to the hidden registers category, while general purpose registers (GPRs) and special purpose registers (SPRs) are user accessible registers. Starting from the UAR group, the table shows how all GPRs have been tested, while only a small minority of SPRs is left undetected. When talking about UAR faults, it is also worth mentioning how many fall within the single-cycle and multi-cycle groups. Concerning STL1, out of all the 4, 359 GPR faults 1, 366 are related to single-cycle instructions and 2, 993 to multi-cycle instructions, while the 2, 232 SPR faults are divided into 2, 219 single-cycle and 13 multi-cycle related faults. STL2, on the other hand, has a total of 23, 814 UAR faults, of which 22, 683 are related to single-cycle instructions and 1, 131 are related to multi-cycle instructions, and the 108 SPR faults can be grouped into 98 single-cycle and 10 multi-cycle related faults. Finally, STL3 has 2, 853 faults of which 1, 367 are related to single-cycle instructions and 1, 486 multi-cycle instructions; of all 47 SPR faults, 11 are single-cycle and 36 are multi-cycle related faults. The distinction between single-cycle and multi-cycle related faults impacts the number of added instructions required to detect the faults as well. As mentioned in Section III-A, single-cycle related faults only need a store instruction to be detected, with an additional overhead of one instruction for SPR faults consisting in moving the value of the special register into a general purpose register so that it can be stored. Multi-cycle related faults, on the other hand, require to duplicate the related multi-cycle instruction and change its operands to make sure that the fault's effects are propagated towards the final cycles of said instruction, plus a store instruction to observe the aforementioned effects at the primary outputs. Most not-detected SPR faults belong to the multi-cycle category, due to the fact that finding the correct operands to propagate the error can be non trivial.

Looking at the HR group, the best results are achieved in the pipeline registers in between the decode and the execute stage, while the other stages pose some challenges. The main reason for having a lower fault coverage stems from the fact that it is not always possible to find the right set of instructions that propagates the values from the pipeline stages to the primary outputs. As for the number of added instructions, experimental data shows that the best results are achieved when adding 5 instructions from the stuck-at fault related test program. It is worth iterating the fact that not every detected fault needs additional instructions, as some faults may cause errors at the same register in the same time instant, thus requiring only one set of added instructions.

V. Conclusions

This work introduces an automated and systematic methodology to detect transition delay faults whose effects have been excited but not observed by already available STLs. Starting from a library of self-test programs developed for stuck-at faults, our approach defines strategies to detect faults based on where their effects propagated and stopped inside the DUT, dividing them into user accessible registers and hidden register groups. Experimental results gathered on a RISC-V test case show that we are able to detect almost every fault affecting UARs, with the worst case scenario being a 98.76% UAR fault coverage. Data on HR faults, on the other hand, show that we are capable of detecting from 5% to more than 15% of all HR faults. Such increase in fault coverage comes with a reasonably small increase of the code size, with the worst case scenario consisting in about 22% added code size for UAR faults, while the contribution for HR faults is practically negligible. The main strength of this work resides in the fact that it is completely automated, hence not requiring any effort from the test engineer, and can drastically enhance the quality of the available STL.

Future works will include the refining of strategies to test HR faults, in order to match as closely as possible the upper bounds in recoverable fault coverage presented in [21].

References

[1] P. Bernardi *et al.*, “On-line functionally untestable fault identification in embedded processor cores,” in *Design, Automation & Test in Europe Conference Exhibition (DATE)*, 2013.

[2] M. Psarakis *et al.*, “Systematic software-based self-test for pipelined processors,” in *ACM/IEEE Design Automation Conference (DAC)*, 2006.

[3] ——, “Microprocessor Software-Based Self-Testing,” *IEEE Design & Test of Computers*, 2010.

[4] N. Hage *et al.*, “On Testing of Superscalar Processors in Functional Mode for Delay Faults,” in *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*, 2017.

[5] A. S. Oyeniran *et al.*, “Implementation-Independent Functional Test for Transition Delay Faults in Microprocessors,” in *Euromicro Conference on Digital System Design (DSD)*, 2020.

[6] K. Christou *et al.*, “A Novel SBST Generation Technique for Path-Delay Faults in Microprocessors Exploiting Gate- and RT-Level Descriptions,” in *IEEE VTS*, April 2008.

[7] C. H. . Wen *et al.*, “On a software-based self-test methodology and its application,” in *IEEE VTS*, 2005.

[8] V. Singh *et al.*, “Instruction-Based Self-Testing of Delay Faults in Pipelined Processors,” *IEEE Transactions on VLSI Systems*, Nov 2006.

[9] P. Bernardi *et al.*, “Development Flow for On-Line Core Self-Test of Automotive Microcontrollers,” *IEEE Transactions on Computers*, 2016.

[10] Wei-Cheng Lai *et al.*, “Test program synthesis for path delay faults in microprocessor cores,” in *IEEE ITC*, 2000.

[11] P. Bernardi *et al.*, “A Deterministic Methodology for Identifying Functionally Untestable Path-Delay Faults in Microprocessor Cores,” in *International Workshop on MTV*, Dec 2008.

[12] M. Grosso *et al.*, “Software-Based Self-Test for Transition Faults: a Case Study,” in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2019.

[13] R. Cantoro *et al.*, “In-field functional test of can bus controllers,” in *IEEE VTS*, 2020.

[14] A. Apostolakis *et al.*, “Test Program Generation for Communication Peripherals in Processor-Based SoC Devices,” *IEEE Design & Test of Computers*, vol. 26, no. 2, 2009.

[15] A. van de Goor *et al.*, “Memory testing with a RISC microcontroller,” in *DATE*, 2010.

[16] Hitex, “Microcontroller self-test libraries.” [Online]. Available: https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetlib/

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

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

[19] 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

[20] J. Perez Acle *et al.*, “Observability Solutions for In-Field Functional Test of Processor-Based Systems,” *Microprocessors and Micros.*, 2016.

[21] R. Cantoro *et al.*, “Self-test libraries analysis for pipelined processors transition fault coverage improvement,” in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.

[22] C. Y. Chen *et al.*, “Reinforcement-Learning-Based Test Program Generation for Software-Based Self-Test,” in *IEEE Asian Test Symposium (ATS)*, 2019.

[23] A. Ruospo *et al.*, “On-line Testing for Autonomous Systems driven by RISC-V Processor Design Verification,” in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2019.

[24] A. Jasnetski *et al.*, “On automatic software-based self-test program generation based on high-level decision diagrams,” in *IEEE LATS*, 2016.

[25] ——, “Automated software-based self-test generation for microprocessors,” in *International Conference MIXDES*, 2017.

[26] A. Riefert *et al.*, “A flexible framework for the automatic generation of sbst programs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, 2016.

[27] ETH Zurich and Università di Bologna, “PULPino microcontroller system.” [Online]. Available: https://github.com/pulp-platform/pulpino

[28] Silvaco, “Silvaco 45nm open cell library.” [Online]. Available: https://www.silvaco.com/products/nangate/FreePDK45_Open_Cell_Library/