Development Flow for On-Line Core Self-Test of Automotive Microcontrollers

Paolo Bernardi, *Member, IEEE*, Riccardo Cantoro, *Student Member, IEEE*, Sergio De Luca, Ernesto Sánchez, *Senior Member, IEEE*, and Alessandro Sansonetti

Abstract—Software-Based Self-Test is an effective methodology for devising the online testing of Systems-on-Chip. In the automotive field, a set of test programs to be run during mission mode is also called Core Self-Test library. This paper introduces many new contributions: (1) it illustrates the several issues that need to be taken into account when generating test programs for on-line execution; (2) it proposed an overall development flow based on ordered generation of test programs that is minimizing the computational efforts; (3) it is providing guidelines for allowing the coexistence of the Core Self-Test library with the mission application while guaranteeing execution robustness. The proposed methodology has been experimented on a large industrial case study. The coverage level reached after one year of team work is over 87 percent of stuck-at fault coverage, and execution time is compliant with the ISO26262 specification. Experimental results suggest that alternative approaches may request excessive evaluation time thus making the generation flow unfeasible for large designs.

Index Terms—Microprocessors and microcomputers, reliability and testing, software-based self-test

1 Introduction

The diffusion of electronic systems in the automotive field is increasing at a fast pace. Car makers constantly demand from electronic manufacturers for faster, less expensive, less power-consuming, and more reliable devices. Microprocessor-based systems are employed in cars for a great variety of applications, ranging from infotainment to engine and vehicle dynamics control, including safety-related systems such as airbag and braking control.

The use of microprocessor systems in safety- and mission-critical applications, calls for total system dependability. This requirement translates in a series of system audit processes to be applied throughout the product lifecycle. Nowadays, some of these processes are common in industrial design and manufacturing flows. These include risk analysis, design verification and validation, performed since the early phases of product development, as well as various test operations both during and at the end of manufacturing or assembly steps. Increasingly often, additional test operations need to be applied during the product's mission life, such as periodic online testing and concurrent error detection. The reliability requirements are met by trading off fault coverage capabilities with admissible implementation costs of the selected solutions.

Within the scope of microprocessor-based integrated systems, the Software-Based Self-Test (SBST) approach has been addressed for a long time by different teams in the research community [1], [2], [3]. SBST techniques basically consist in letting the CPU running a sequence of code words to excite and propagate errors affecting the circuit [4]. Compared to traditional hardware-based test solutions, such as Built-In Self-Test (BIST), it presents many advantages, such as: the possibility of autonomously testing [5] and diagnosing [6] both the microprocessor and the controllable peripherals; working in normal mode of operation; not requiring any hardware modifications; allowing at-speed test application (i.e., at the circuit nominal frequency). Nevertheless, SBST methodologies have their applicability in industry limited by the difficulty to write efficient and effective test programs and to devise suitable methodologies for test application.

While in the manufacturing test arena, BIST solution are chosen because these solutions achieve high coverage in a short time and with a simple development flow, regarding online test application, SBST is the preferred solution for periodically monitoring the system health without interfering with the normal mission behavior [7], [8]. A standard solution adopted by the industry is the periodic application of test from a Core Self-Test (CST) library composed of SBST programs. In such an approach, the microprocessor is periodically forced to execute a self-test code able to detect the possible occurrence of permanent faults in the processor itself and the peripherals connected to it (Fig. 1). Such procedures are designed to activate possible faults, then compress and store the self-test results in an available memory space, or raising a signal when the test has not ended correctly.

As far as test program generation is concerned, many approaches can be found in the literature, employing manual or automated approaches, which are suited to target different processor architectures and fault models as described in [4]. However, setting up an efficient CST development

• *P. Bernardi, R. Cantoro, and E. Sánchez are with the Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy. E-mail: {paolo.bernardi, riccardo.cantoro, ernesto.sanchez}@polito.it.*
• *S. De Luca and A. Sansonetti are with STMicroelectronics, Agrate Brianza, Italy. E-mail: {sergio.deluca, alessandro.sansonetti}@st.com.*

Manuscript received 1 Feb. 2015; revised 29 July 2015; accepted 17 Sept. 2015. Date of publication 5 Nov. 2015; date of current version 10 Feb. 2016. Recommended for acceptance by C. Bolchini, S. Kundu, and S. Pontarelli. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2015.2498546

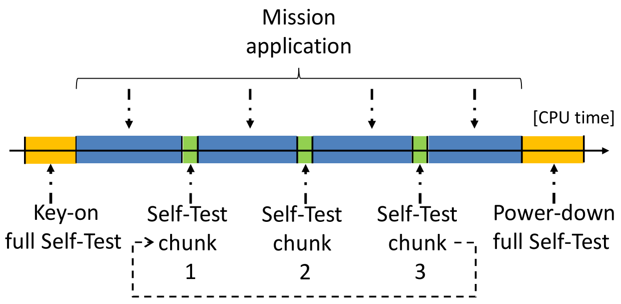

Fig. 1. On-Line Self-Test application example.

flow requires solving a number of additional issues regarding test program generation, organization, and grading [9]. Coping with such online requirements means both to introduce additional rules to be respected along test program generation, and to include additional code parts that may impact on the CST development time.

This paper presents an innovative and comprehensive approach for the development of a CST library for microprocessors in safety-critical automotive embedded systems. Such a CST library can be integrated in the Operating System installed on-chip (it will be referred to as OS), i.e., which is responsible for running the mission application. The pursued goal is to satisfy all reliability requirements imposed by emerging standards such as ISO 26262 [10], which require a constant monitoring for the possible occurrence of permanent faults in the circuit along its mission life.

The technical content of the paper deals with the most relevant aspects of on-line test programs characteristics and their development flow. Specifically, the paper progresses the state-of-the-art by describing and discussing:

1. the constraints to be taken into account when generating self-test programs to be run on-line;
2. the requirement for a robust on-line execution of self-test programs in coexistence with an embedded operating systems;
3. an effective development flow organization aiming at minimizing the computational efforts.

Moreover, the paper shows the results that have been collected on an industrial case study during one year. The impact of online requirements is evaluated on a large 32-bit microprocessors embedded in an automotive System-on-Chip (SoC) manufactured by STMicrolectronics. Code overheads and adaptation toward on-line of the generation strategies are reported; experimental results also demonstrate that the development of a CST becomes unfeasible on processors with a significant dimension, without careful planning for proper resource partitioning and order in the CST creation.

The rest of the paper is organized as following: Section 2 describes the on-line constraints that are encountered while generating test programs. Section 3 details the characteristics required to guarantee robustness and full compliancy with the OS. Section 4 illustrates an effective development flow suitable for large microcontrollers. Section 5 reports experimental results, and Section 6 concludes the paper.

2 Core Self-Test Generation Constraints

SBST is widely perceived as a proper method for accurate and non-invasive autonomous test. In a few words, a test program runs and eventually detects misbehavior by simply exercising the processor functionalities. This process intrinsically respects power constraints, since the test programs are executed under the same conditions of the mission mode. SBST does not ask for additional test circuitries, and is quite cheap in terms of features and commodities required to the test equipment.

When dealing with a (CST), which has to be applied online, the test programs have to share processor resources with the mission application, i.e., the OS which is managing mission's tasks; this coexistence introduces very strong limitations compared to manufacturing tests:

1. CST programs need to be compliant with a standard interface, enabling the OS to handle them as normal processes. This interface must guarantee processor status preservation and restoration, even in case of higher priority requests (e.g., preemption).
2. The CST programs need to be generated following execution time constraints, due to the resources occupation that can be afforded by the mission environment. In particular, this is strictly required when a test cannot be interrupted because using critical resources (e.g., special purpose registers).
3. There is a strong limitation in terms of memory resources, due to the mission code and data characterizing the OS. To face this issue, it is usually recommended to:
   - provide the CST as a set of precompiled programs stored in binary images to be run along mission mode, possibly scheduled and loaded by the operating system;
   - not to refer to any absolute addresses when branching, as a result, the test code can be stored in the memory, copied, and launched from any location without any functional or coverage drawback;
   - not to refer to absolute addresses when accessing the data memory;
   - identify possible memory constraints from the point of view of the OS restrictions, and indispensable locations to be reserved for test purposes.

Moreover, targeting effort reduction, the test should be created also taking into account the characteristics of the general processor family, in order to reduce code modifications when transferring the CST library to another processor core belonging to the same family architecture. The next paragraphs face these questions and provide some guidelines for easily taking early decisions.

On-line execution constraints may in same cases limit the effectiveness of the usual methodologies for writing test programs, which has to be rethought to fit to the real system.

3 Core Self-Test Execution Management

As briefly described in the introduction, the inclusion of SBST routines in the mission environment is a critical issue. To face the problematic aspects of this integration, after the generation, we consider three major points related to test program execution:

- cooperation with other software modules, usually related to the mission environment such as the OS;

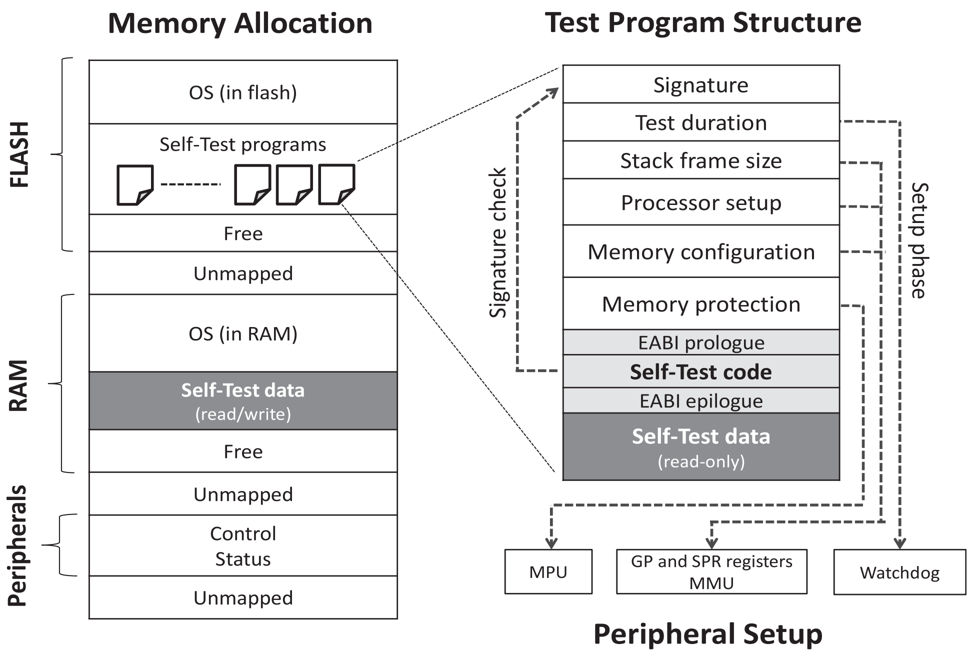

Fig. 2. Test program encapsulation and loading for execution phase.

- context switching and result monitoring;
- robustness in case of faulty behavior, which is strictly related to interruption management.

3.1 Test Encapsulation

Considering the cooperation with other software modules, such as the threads launched by the OS, the test program suite needs to be constructed by including key features enabling the test to be launched, monitored and eventually interrupted by higher priority processes of the mission management system.

The test program has to be carefully structured, in order to configure memory areas and peripheral resources for test purposes, as also graphically depicted in Fig. 2. All the test programs are normally stored and executed in the flash memory.

First of all, in order to be compliant with the mission software environment, a viable and strongly suggested solution for the development of the OS and software modules for automotive embedded microcontrollers is the adoption of the Embedded-Application Binary Interface (EABI). Widely used EABIs include PowerPC [11], ARM EABI2 [12], and MIPS EABI [13]. EABIs specify standard conventions for file formats, data types, register usage, stack frame organization, and function parameter passing of a software program. Thus, every test program includes an EABI prologue and epilogue (Fig. 2), in charge of saving and restoring the mission status.

The EABI frame is created by the test code as soon as it begins. Thus, any scheduler can launch the test execution, e.g., the scheduler available in the OS hosting the test routine. Moreover, we propose the inclusion of extra information which can allow a test scheduler to perform a proper running environment setup.

Such additional information, or metadata, encompasses:

- stack frame size;
- special purpose register setup;
- memory protection setup;
- test duration.

Metadata are used by the test program during different phases, some at the setup:

1) duration time (i.e., watchdog setup);
2) stack frame size (i.e., space available for mission configurations to be saved and local variables of the test program);
3) processor setup (i.e., special purpose register ad-hoc values);
4) memory configuration (i.e., virtual memory initialization);
5) memory protection (i.e., to manage wrong memory accesses through exceptions); and others at the execution end:
6) signature (i.e., test results check).

Such a memory structure can be also stored in the mass memory until it is loaded to be run from any portion of the available memory, according to the features already described in Section 2 (i.e., relocation).

3.2 Context Switching to Test Procedure

Test programs structured as described Section 3.1 can be integrated into any mission OS as normal system tasks. Proper context switching is managed by the EABI interface; additional setup may be required, according to the characteristics of the test program, and it is managed internally by the test programs themselves, by means of the additional metadata.

We identify three general cases, each one demanding for proper metadata to be used in setup procedures:

- *run-time tests:* can be interrupted by mission requests; usually used to cover computational modules such as arithmetic modules;

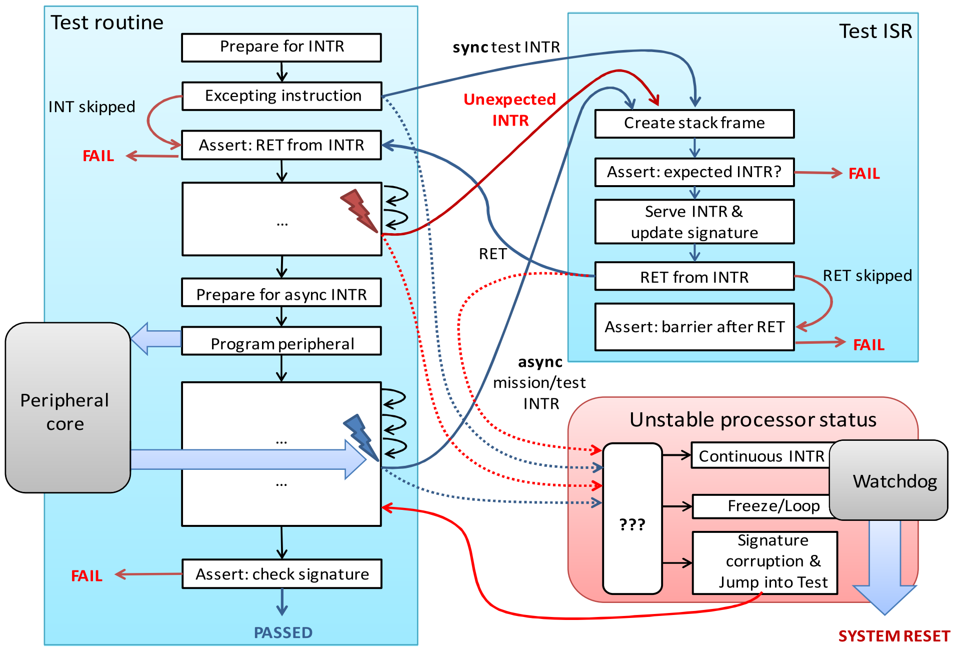

Fig. 3. Expected and unexpected exception management scenario.

- *non-exceptive tests:* require the manipulation of SPR register, such as for testing the Register File;
- *critical tests:* intentionally raise interrupts and make use of peripheral cores, such as a procedure for testing modules managing software exceptions.

Run-time tests are the easiest to manage. They only require creating a stack frame according to EABI compliancy; stack frame size is minimal. It is suggested to execute this kind of tests with low privileges, i.e., user mode, because they shall never request interruptions or privileged instruction execution in the fault-free scenario. No other special setup is required. EABI compliancy can be satisfied during the execution, and other OS threads can preempt the test execution.

Non-exceptive tests are slightly less easy to manage because these require resources that are not allowed to be directly used in the EABI context. For this category, additional test setup steps have to be executed before running:

- disable the external interrupts to avoid preemption;
- save all special and general purpose registers in a larger stack frame memory area;
- modify the content of special and general purpose registers according to the processor setup information.

Moreover, some closing operations are needed at the conclusion of the test execution to restore the initial configuration. During the execution of these tests, no preemption is allowed because the compliancy with the EABI standard cannot be guaranteed.

When considering *Critical tests*, other than saving-restore all registers and disable external interrupt sources, more restrictive requests have to be accomplished:

- the Interrupt Vector Table (IVT) and the related registers in case an alternative IVT is required for testing purposes;
- the current status and control registers of the used peripheral modules, such as the interrupt controller configuration and the MMU.

3.3 Interruption Management and Robustness

Interrupt mechanisms, which are managing synchronous and asynchronous exceptions, need to be handled with extreme care, because they are not only intentionally raised for testing purposes.

We discriminate three types of exception:

- intentionally provoked exceptions, i.e., to test processor exceptions;
- unexpected, induced by an erroneous execution that is provoked by a faulty configurations;
- mission mode interruptions.

Intentionally provoked interruptions can be both synchronous or asynchronous. Situations like system calls, illegal memory access, illegal instructions and privilege related operations are synchronous, as they have to be forced by the code itself. On the other hand, asynchronous interruptions are raised by peripheral cores.

To test exceptions it is therefore necessary to both rise and manage them (Fig. 3). If the circuitries managing the interrupt have not been corrupted by a fault, each single forced exception is correctly managed, meaning that a test Interrupt Service Routine (ISR) is accessed. Such an ISR is configured at the time when the scheduler prepares the environment for the test program execution and replaces the mission ISR with the test ISR.

The code included in the test ISR is also used to add significant contents into the signature, e.g., the processor's status registers. In presence of a fault, the execution flow may be altered and an exception intentionally scheduled but is not raised. In this case, the signature is not updated by the test ISR, and the right signature value is not produced.

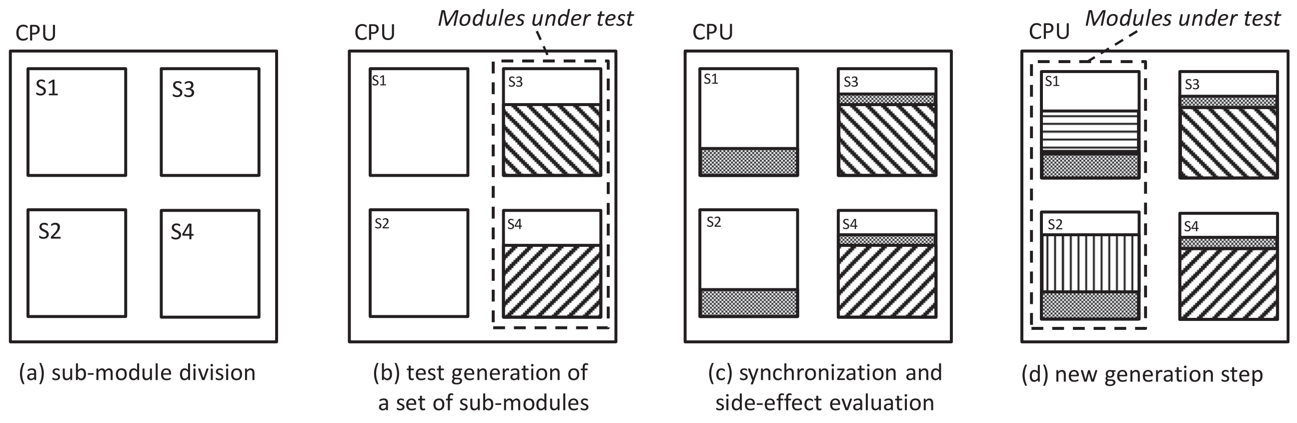

Fig. 4. Sub-module identification and visualization of the coverage figure evolution along the proposed generation steps.

Furthermore, the exception management is also crucial for detecting faults producing execution flow deviations that lead to an unexpected processor's internal status or cause unexpected synchronous interruptions. Typical cases are a legal instruction wrongly decoded as an illegal instruction format, or an illegal memory access raising an interruption by the memory protection unit mechanism. If this situation occurs during the execution of any test program, the test ISR should ideally be able to recover such an erroneous (due to the fault) execution flow and to record the wrong behavior observed. Some countermeasure can be adopted to identify unexpected interrupt requests, such as performing an assertion in the ISR prologue to check a password stored into a GPR before the interrupt is intentionally raised. A similar method is implemented for checking the correct return from interrupt, e.g., by completing the test execution with an assertion.

This technique is making the test code quite robust, but more work is needed if the processor status become unstable, resulting in spurious and repeated exceptions or infinite loops. In the latter case, an external mechanism have to be implemented in order to bring the system into a safe status, i.e., by watchdog timer.

These cases are graphically shown in Fig. 3, where solid lines are showing expected interrupts while dashed are showing the effect in case the processor status is unstable. During the execution of run-time test programs, mission interrupts need to be identified and served as soon as possible, i.e., passed to the mission OS. By following the EABI standard, it permits to easily manage this case.

4 Core Self-Test Development Flow

The major cost in the development flow of a CST is the computational effort required to generate the test program suite: the fault grading process [9], which evaluates the effectiveness of a test program (by assessing the fault coverage), represents a severe bottleneck. Moreover, the cost for developing the test program infrastructure described in the previous paragraphs and required by the on-line execution is not negligible. For instance, for a medium sized embedded processor with about 200 k stuck-at faults, fault simulating 1ms program may require some three days on a 2 GHz quad-core workstation at the maximum parallelism allowed.

Such a cost becomes unsustainable if the generation process is iterative [18] and produces many programs before achieving a good coverage. We propose a methodology able to achieve a significant optimization of development time and resources based on the following principles:

- *modularity*: the embedded processor is not tackled as a unique module, but its sub-modules are considered separately (e.g., ALU, CTRL Unit, etc.); this means that the processor's fault universe is selectively divided into several smaller fault lists for being effectively attacked while generating the test programs;
- *parallelization*: by facing modules separately, it facilitates the distribution of the development process on the available workstations/test-engineers (see more details in 4.1);
- *side-effect*: by developing a test for a specific sub-module, it is likely to have a side-effect that is the coverage of faults belonging to other sub-modules.

The strategy we propose is based on these principles and consists on the iterated execution of two steps until all sub-modules are considered:

1. to generate, possibly in parallel, the test programs for a set of carefully selected sub-modules until these are sufficiently covered (more details in 4.2);
2. to perform a synchronization among all the previously generated test programs, to evaluate their effectiveness over a larger fault list; to synchronize means to fault grade test programs that are generated for a specific sub-module over a different (larger) list of sub-modules.

In order to better explain the proposed flow, we can consider a simple example of a microprocessor's CPU, which is modularized into four sub-modules: *s1*, *s2*, *s3*, and *s4* (Fig. 4a).

In the first step (Fig. 4b), two modules (*s3* and *s4*) are considered in parallel and graded separately, i.e., during the generation process for module *s3* only its own faults are considered.

As soon as the coverage of these sub-modules is satisfactory, the generated test programs are graded over the other parts of the CPU (Fig. 4c); this synchronization step brings to observe a positive side-effect on the coverage of modules *s1* and *s2*, as well as on *s3* when grading the test program for *s4*, and vice versa.

A new generation step is then started on *s1* and *s2* (Fig. 4d); it is worth to mention that the previous steps were beneficial because the starting fault lists of *s1* and *s2* have been lightened before facing their generation process.

The main advantage of using this approach is a faster development of the test suite because:

- the fault lists to be considered are significantly smaller than the complete fault list, resulting in a faster fault grading that usually takes in the order of minutes to complete;
- after each synchronization step, the number of active faults in new sub-modules is reduced, again leading to a speed-up of the fault simulation process.

4.1 Resources Partitioning

As briefly stated in the previous paragraph, the processor is divided into sub-modules to consider *independent* fault lists in parallel, and their selection is the major issues in the preliminary phase preceding the generation effort.

For being *independent*, fault lists need to be

- *non-overlapping:* one fault has to belong to only one fault list;
- *functionally orthogonal:* faults in the same independent fault list need to belong to modules related to one specific functionality.

The non-overlapping criterion dictates that the same fault must be considered only one time in the process; on the other hand, when dealing with orthogonality, a fault must be included in the most relevant fault list from the point of view of the functionality of the related gate.

The process of selecting the set of fault lists is not trivial. In our flow, this process relies on:

- manuals and documentation of the microcontroller with specific indications about the micro-architecture;
- hierarchy of the microcontroller netlist;
- test engineer expertise.

To identify independent fault lists, it requires:

- analysis of the processor functionalities;
- mapping of the functionalities over the microcontroller hierarchical netlist.

Most of the fault lists directly derive from a specific module, but it is frequent that different sub-modules are squeezed into a single fault list if related to the same processing functionality. As an example, the faults of a multiplication unit usually constitute an independent fault list. On the contrary, there are several multiplexers that seem to be independent netlist modules, but actually compose the feed-forwarding logic in the processor's pipeline; thus, faults belonging to these multiplexers have to be grouped into the same fault list, which is functionally orthogonal and non-overlapping with other modules.

Concerning computational resources allocation, once the independent fault lists have been identified, the following rules aim at maximizing the number of fault lists to be considered in parallel:

- fault coverage calculation for sub-modules may be allocated on a single or many threads according to:
  - number of available threads per CPU;
  - number of EDA tool licenses;
  - fault list size;
- fault coverage calculation for synchronizing the results of multiple test programs among multiple sub-modules should be allocated on many threads.

4.2 Optimized Test Programs Generation Order

Based on the side effect principle described above, it is crucial to select the most promising order to proceed in the test program generation. The decision needs to be tailored on the specific architecture under analysis.

In the following, we propose general guidelines for determining the generation order considering the most common microprocessor architectures used in automotive. In our development flow, we organize the generation order according to horizontal and vertical flow rules. Vertical flow rules demand to split the flow into consecutive *levels*, such that by testing all the modules into a given level, a large positive side-effect in terms of fault coverage is observed when moving to the next level.

We are currently proposing to divide the flow into levels according to the following rules:

1) to consider first those units that can be mapped on specific assembly instructions or specific architectural programming mechanisms;
2) to continue with memorization and control flow resources;
3) to conclude with modules which functionalities are transparent to the programmer.

According to the presented strategy, a synchronization step is needed after completion of the currently considered level before moving to the next; this synchronization step involves all sub-modules of the next level, i.e., to reduce the size of the fault lists to be considered successively. By looking at the problem in a horizontal manner, it is also possible to identify many parallel *branches* which are still complying with vertical requirements. This horizontal view consists in identifying branches, so that a negligible side-effect crosses branches belonging to different horizontal views.

Based on the previous rules, for a typical automotive-oriented architecture, we individuate a development flow based on three levels and two branches (Fig. 5). However, additional branches can be added when considering microprocessors equipped with special features, e.g., caches [14], shared memory schemes [15], and Floating-Point unit [16].

We suggest considering two broad categories of sub-module as first levels:

level 1 – branch A) ALU: easy to test, sub-modules that ask for the execution of specific arithmetic and logical instructions. Side effect will be maximized towards the REGISTER FILE by an accurate selection of registers to be used as operands and in the control flow management.

level 1 - branch B) SPECIAL: sub-modules that encompass Exceptions Management, Branch Prediction, and Virtual Memory-related modules, e.g., the Memory Management Unit (MMU) module. These sub-modules are hard to cover, requiring specific instructions and sequences of instructions. They will produce a very large positive side-effect on ADDRESS-related modules.

Once a sufficient coverage on these sub-modules is reached, it is suggested to proceed in a synchronization

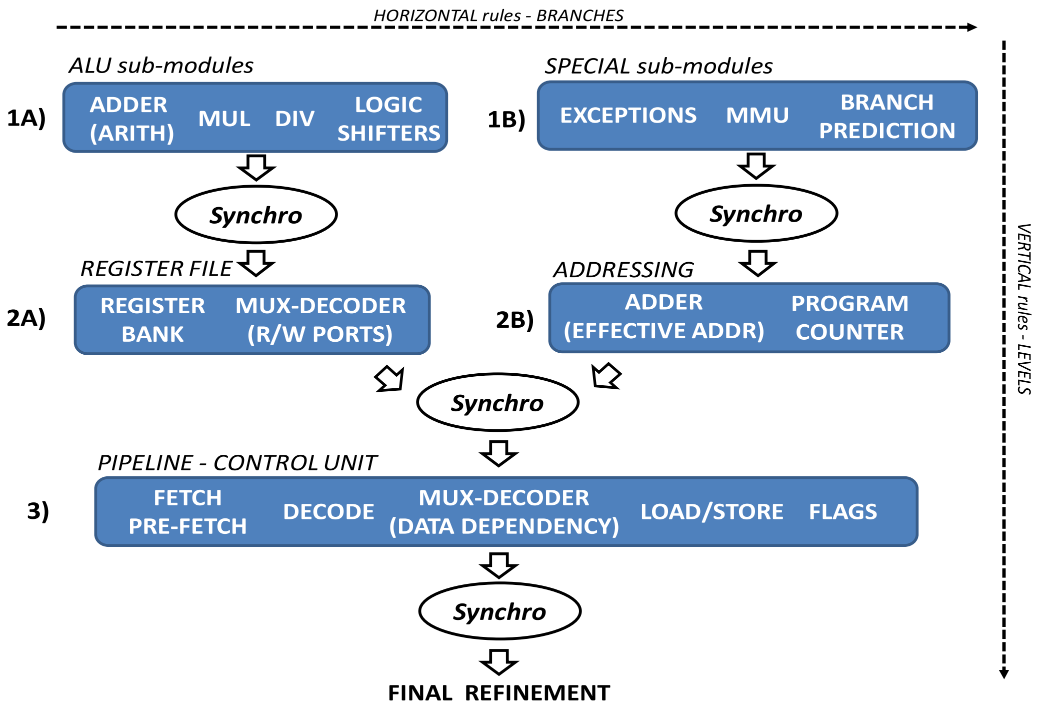

Fig. 5. Proposed test program development order organized in levels and branches, and synchronization steps.

process. The set of programs developed for 1A) are evaluated on the REGISTER FILE fault lists, while 1B) is graded on the ADDRESS-related modules. As a result, the number of active faults to be then considered will be greatly reduced.

level 2 – branch A) REGISTER FILE: the test of the register file module is straightforward, being many papers describing effective sequences to test. In the proposed generation method, it is suggested to reorder instructions and operands in order to induce the usage of DATA DEPENDENCY structures in the PIPELINE.

level 2 – branch B) ADDRESSING: by having completed 1B), the most of the faults included in the ADDRESS-related modules, such as Branch unit, Effective Address calculation, and Program Counter, will result as already covered. This step is therefore a completion of the previous one, which is done mainly by adding memory operation and branches to specific addresses.

A synchronization step is then operated on the PIPELINE and CONTROL UNIT modules, followed by **level 3** with no more branches, which consists on an ad-hoc generation step for these level modules.

To complete the process, the entire test suite obtained along this process is evaluated on the whole processor fault universe, eventually by adding refinement programs to cover corner cases and specific configurations not considered along the previous steps.

5 An Industrial Case-Study

The methodology herein introduced has been applied to a SoC including a 32-bit pipelined microprocessor based on the Power Architecture. The SoC is employed in safety-critical automotive embedded systems, such as airbag, ABS, and EPS controllers and is currently being manufactured by STMicroelectronics.

The microcontroller's cost-efficient processor core is built on the Power Architecture technology and designed specifically for embedded applications. The processor integrates a pair of integer execution units, a branch control unit, instruction fetch unit and load/store unit, and a multi-ported register file capable of sustaining six read and three write operations per clock cycle. Most integer instructions execute in a single clock cycle. Branch target prefetching is performed by the branch unit to allow single-cycle branches in many cases. It contains a Memory Management Unit (MMU) and a Nexus Class 3 module is also integrated for external debug purposes.

The 32-bit processor utilizes an in-order dual-issue five-stage pipeline for instruction execution. These stages are:

- Instruction Fetch (stage 1)
- Instruction Decode/Register file Read/Effective Address Calculation (stage 2)
- Execute 0/Memory Access 0 (stage 3)
- Execute 1/Memory Access 1 (stage 4)
- Register Write-Back (stage 5)

The stages operate in an overlapped fashion, allowing single clock instruction execution for most of the available instructions.

The integer execution unit consists of a 32-bit Arithmetic Unit (AU), a Logic Unit (LU), a 32-bit Barrel shifter (Shifter), a Mask-Insertion Unit (MIU), a Condition Register manipulation Unit (CRU), a Count-Leading-Zeros unit (CLZ), a 32 $\times$ 32 Hardware Multiplier array, and result feed-forward hardware. Integer EU1 also supports hardware division. Most arithmetic and logical operations are executed in a single cycle with the exception of multiply, which is implemented with a two-cycle pipelined hardware array, and the divide instructions. A Count-Leading-Zeros unit operates in a single clock cycle. Two execution units are provided to allow dual-issue of most instructions. Only a single load/

TABLE 1
SBST Strategies Used along the Generation Process

Sub-module	Technique	References
Arithmetic adders Division unit Logic unit Multiplication unit Shifters	Deterministic + Constr. ATPG + Evolutionary	[18], [19], [20], [21]
Exceptions	Manual	[22]
Branch Unit	Deterministic	[23]
Timers	Manual	
Register bank	Deterministic	[24]
Register ports	Deterministic	[24]
EA adder Load store unit program counter	Loop-based + Evolutionary + Manual	[25]
Forward unit	Deterministic	[26]
Decode unit Status/control flags	Deterministic	[27]
Fetch unit	Deterministic	[28]

store unit is provided, and only a single integer divide unit is provided, thus a pair of divide instructions cannot issue simultaneously.

5.2 Experimental Results

During one year of team work, we collected results of a wide number of modules. To test processor through software is a deeply explored field; therefore, in many cases the technique utilized has been borrowed from the literature and adapted to cover the specific modules of the considered processor core. Table 1 reports the list of generation techniques employed to achieve the high fault coverage of each processor sub-module. On selecting these techniques, we resort to some of the most important proposals regarding test program generation available in today's literature.

Concerning automatic approaches, we resorted to both *ATPG*-based techniques, and optimization techniques based on *Evolutionary Algorithms* [17]. Some other techniques (labeled as *Deterministic*) refer to available solutions that exploit the sub-module regularity in order to propose a well-defined test algorithm. Finally, rows labeled as *Manual* refer to pure manual strategies performed by the test engineer exploiting the processor user manual, the ISA, as well as the available HDL processor descriptions.

As stated in Section 2, both the duration and the size of each test program is an online requirement that may vary depending on the mission application and physical limits of the microcontroller (e.g., the memory space available). In our case study, the limitations were given both in terms of duration of single programs, and of overall occupation of the complete test suite. In particular, the maximum duration of a single program labeled as *run-time test* should not exceed 512 clock cycles, while the FLASH memory area reserved for test purposes was limited to 256 kB.

To match these constraints, every method needs to be tailored opportunely:

- *ATPG*-based generation methods can be constrained by asking the automatic engine for high compression and limiting the generation to a maximum number of patterns; the generated patterns may be eventually transformed into many test programs compliant with duration constraints;
- fitness values used in the *Evolutionary* computation experiments include program size and length measurement; in such a way, the programs exceeding the imposed limitations can be discarded;
- *Deterministic* and *Manual* require additional efforts by the test engineer to fit the programs length and size; more easily, if too long, they can be split into several shorter programs.

Code characteristics fitting on-line requirements were also considered in all cases, such as having relocatable code (absolute branches and access by absolute address to memory locations were not allowed) and resorting to limited portion of memory space (1 kB) reserved for test.

As described in Section 3, each generated program is encapsulated into the EABI standard frame and includes the additional code sequences that guarantee the test robustness.

For the current case of study, the EABI compliant frame is accounting for very few instructions at the beginning of procedures (e.g., three-five instructions); this number increases whereas:

- extra registers have to be saved before being used and finally restored to their original values (e.g., non-volatile registers, or special purpose registers, such as the Microprocessor Status Register);
- memory resources need to be protected (e.g., Memory Protection Unit is exploited);
- peripheral microcontroller's resources need to be programmed for test robustness (e.g., watchdog timers, performance counters).

The number of additional instructions required to afford robustness other than raw compliancy with EABI standards are about 20 instructions. Additionally, to further enforce robustness, additional instructions were added when a context switching is forced via exception. Concerning the development flow, the fault list generation and the adopted generation order follows the generic indications provided in Section 4.

The fault lists were generated mainly according to the processor functionalities which are directly related to specific modules in the netlist hierarchy. There are some exceptions, due to the fact that the considered microcontroller is dual-issue and replicated arithmetic modules, such as the adders, were grouped together in a single fault list; in a different way, the data-forwarding unit is composed of several multiplexers, which faults are jointly considered. Another interesting case of resource partitioning is related to the multi-port register file that is contributing with two fault lists, the register bank and the register ports (decoders and multiplexers); this is due to both the fault list size that we need to split, and the different functionalities.

Table 2 shows the evolution of the coverage along the development flow. The final fault coverage is 87.23 percent on the fault list that includes around 750 k stuck-at faults.

For different reasons, there are modules not well covered:

- modules managing exceptions: because it is not possible to purposely exercise all of them (e.g., it is not possible to force a bus error which asks the exception unit to intervene);

TABLE 2
Coverage Evolution along the Development Flow

Sub-module	#faults	Single 1A	Synchro 1A	Single 1B	Synchro 1B	Single 2A	Single 2B	Synchro 2A+2B	Single 3	Synchro 3
		FC [%]	FC [%]	FC [%]	FC [%]	FC [%]	FC [%]	FC [%]	FC [%]	FC [%]
Arithmetic adders	5,996	**95.03**	97.93	–	–	–	–	98.27	–	98.52
Divider	19,018	**83.98**	83.98	–	–	–	–	83.99	–	84.82
Logic instructions	22,294	**76.32**	78.57	–	–	–	–	78.70	–	83.34
Multiplier	78,094	**91.18**	92.62	–	–	–	–	92.62	–	95.90
Shifters	14,172	**87.95**	92.96	–	–	–	–	93.97	–	96.32
Exceptions	40,718	–	–	**66.08**	67.17	–	–	68.16	–	72.48
Branch prediction	24,489	–	–	**70.91**	70.95	–	–	72.67	–	72.67
Timers	7,683	–	–	**88.21**	88.43	–	–	88.46	–	89.70
Register bank	83,764	–	71.21	–	–	**84.15**	–	89.38	–	92.66
Register ports	126,329	–	69.17	–	–	**94.93**	–	97.67	–	98.09
Program counter	26,060	–	–	–	66.07	–	**68.66**	69.42	–	70.09
EA adder	5,228	–	–	–	66.51	–	**92.02**	93.75	–	94.57
Fetch unit	71,582	–	–	–	–	–	–	69.45	**82.39**	83.54
Forward unit	84,758	–	–	–	–	–	–	70.95	**84.29**	84.82
Status flags	33,277	–	–	–	–	–	–	59.31	**78.08**	78.61
Control flags	10,328	–	–	–	–	–	–	64.21	**66.83**	69.83
Decode unit	62,876	–	–	–	–	–	–	50.12	**92.46**	93.08
Load/Store unit	15,971	–	–	–	–	–	–	73.50	**75.42**	76.73
Glue logic	19,425	–	–	–	–	–	–	–	–	63.36
TOTAL	*756,789*	–	*36.07*	–	*9.74*	–	–	*76.87*	–	*87.23*

- branch prediction, program counter, and load/store units: due to the memory mapping configuration of the specific SoC, not all bits in the addressing registers can be functionally touched;
- processor status and control flags: since many of these flags cannot be used because controlling circuitries outside the processor core.

As a complement to the fault coverage measurements, the dimension and duration and coverage of the test along the entire development flow are included in Table 3. Having a frequency of working of 150 MHz, the overall time required for executing all 73 tests is about 0.8 ms.

It is interesting to note, how the synchronization phases produce a very strong positive cascade effect over the modules not yet considered; at least the half of the faults of the modules that are going to be considered during the next generation steps were pruned from the list without any additional effort. Table 2 also permits to remark that the synchronization steps cause coverage improvement also for modules of the current and previous levels of the same branch, as described in Section 4.2.

A significant advantage in terms of grading time reduction is achieved by a proper development order which is maximizing the cascade effect. As an example of effectiveness, by adopting the proposed order, the generated test programs over the 139,574 faults of arithmetic modules included in 1A (level 1 – branch A) led to a positive side effects on 2A consisting in 147,029 over 210,093 faults (corresponding to about 70 percent), i.e., these faults are already covered without any specific generation effort for 2A. In other words, the fault simulation experiments carried on level 2A need to consider only 63,064 faults.

To the sake of completeness, we also computed the results obtained by implementing an alternative generation order, considering by first the modules of 2Aand then the ones of 1A. We tackled the 210,093 faults of register bank and ports by obtaining a fault coverage comparable with results in Table 2, and evaluated the side effect of such programs over 1A: only 10,318 faults (or 7.4 percent) were already covered over the total amount of 139,574 faults of the arithmetic modules.

The reduction in the fault list cardinality, achieved by properly ordering sub-modules and synchronizing fault lists, induces a great time gain due to a large reduction of fault simulation efforts. The effect is not limited to successive synchronization, but it permits faster generation iterations as required by evolutionary algorithm.

Table 4 shows the elapsed time for fault simulation in two cases:

1) a raw development flow not using synchronization but simply considering sub-modules separately;
2) a development flow following the proposed order and implementing synchronization between levels.

All the experiments were executed on a single core of a 2 GHz processor; the resulting times would be reduced by running multi-process fault-simulations.

It is worth to notice that the fault simulation time becomes excessive if not implementing synchronization. As well, the development order is important to minimize the fault simulation efforts. Supposing again that level 2 branch

TABLE 3
Number of Test Programs, Duration and Code Size

Development Flow Step	Number of test programs	Duration [Clock Cycles]	Code size [kB]
Single 1A	29	8,840	23
Synchro 1A			
Single 1B	8	19,716	11
Synchro 1B			
Single 2A	10	32,634	36
Single 2B	8	28,212	17
Synchro 2A+2B	55	89,402	87
Single 3	18	26,700	32
Synchro 3	73	116,102	119

TABLE 4
CPU Time Comparison for Approaches without and with Synchronization

	Fault simulation time [hours]		
	1) without synchro	2) with synchro	Saved time
Level 1 Branch A	37		-
Level 1 Branch B	122		-
Level 2 Branch A	217	55	74.7%
Level 2 Branch B	72	23	68.1%
Level 3	630	195	69.0%

A has been considered before level 1 branch A, the saved CPU time for fault simulation is decreased from about 37 to 34 hours, which is a negligible gain if compared to those obtained by the proper ordering.

5.3 Test Deployment during Mission

The suite of test programs resulting from the development phases described above is integrated in an industrial demo project for STMicroelectronics. The project handles the whole test set of programs by means of two software modules, and provides the project integrator with a software Application Programming Interface (API), in order to include them in the mission application:

- ***tests for power-on:*** 44 test program, including *Non-exceptive* and *Critical* tests, scheduled by an ad-hoc software module named Boot Time Self-Test Module (BTSTM);
- ***tests for Run-time:*** 29 *Run-Time* test programs handled by an AUTOSAR 4.0 Complex Driver [29] named CST Library.

Both CST Library and BTSTM provide configuration capabilities at compile time, in such a way that the project integrator can selectively activate all programs or a subset of the entire suite. It is up to the user of the API to choose suitable test combinations and a scheduled execution order to fulfill the safety requirements of the system.

In the devised demo, after the execution of the tests for power-on that takes about 0.7 ms, the run-time tests were scheduled according to some specific requirements for mission integration:

- self-test chunks must be less than 5 μs long;
- self-test interrupts the mission application every 500 μs.

Along mission, using the proposed demo setup, the overall self-test length does not exceed 100 μs and a complete self-test is performed in less than 2 ms. Availability of the mission application is reduced by around 1 percent even though that self-test can be preempted at any time.

Along development, the demo test suite was encompassing several verification and validation stages towards software maturity, which were including embedded documentation of the code by means of special comments (e.g., Doxygen tags [30]) that are parsed by external tool for automatically generating user manuals.

Test programs also provide services for returning test results, i.e., error codes such as AUTOSAR DEM errors and malfunctioning signatures computed by the test programs for successive inspection of failing chips. BTSTM assumes that all the available processor functionalities can be exclusively accessed for testing purposes; on the contrary, CST Library has more restrictive requirements.

For validation sakes, we finally conducted two kinds of experiments emulating the in-field behavior of the system:

1) to verify the fault-free behavior, a sample OS was considered that was intensively triggering mission interrupts while self-test executed at regular intervals as described above; a physical target was programmed with such a complete software environment and left running for several hours, tracing the correctness of the test responses and liveness of the system;
2) to investigate on the robustness in case of faults, a specific fault injection campaign was performed by means of complete simulation (i.e., without fault dropping) in order to classify erroneous behaviors, as previously introduced in Section 3.3:
    a) self-test ends with a wrong signature;
    b) self-test is not ending due to deadlock configuration;
    c) self-test ends with unattended exception management, due to:
        i) illegal instruction execution;
        ii) wrong branches in memory areas protected by MPU configuration.

6 Conclusions

This work proposes a development flow for the effective SBST generation of a library of test programs, also referred as CST, to be run online during the mission of automotive systems.

The paper includes:

- identification of online constraints and implemented solutions;
- resources distribution and generation order for a most efficient and fast test program generation along the various sub-modules of the entire processor;
- execution management of the CST library and robustness of its execution.

As a case of study, the paper reports the final results related to a CST library generated for an industrial 32-bit processor core included in an automotive SoC manufactured by STMicroelectronics; the fault coverage obtained in one year team working is more than 87 percent over around 750 k stuck-at faults.

Acknowledgments

The authors would like to thank for the useful discussion and contributions to this work Thomas Zsurmant, Renato Meregalli, and Giovanni Di Sirio from STMicroelectronics, Matteo Sonza Reorda, Michelangelo Grosso, Lyl Ciganda, and Giovanni Squillero from Politecnico di Torino, Oscar Ballan previously working with STMicroelectronics.

References

[1] S. M. Thatte and J. A. Abraham, “Test generation for microprocessors,” *IEEE Trans. Comput.*, vol. C-29, no. 6, pp. 429–441, Jun. 1980.
[2] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, “Deterministic software-based self-testing of embedded processor cores,” in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2001, pp. 92–96.

[3] C. H. P. Wen, Li. C. Wang, and K.-T. Cheng, "Simulation-based functional test generation for embedded processors," *IEEE Trans. Comput.*, vol. 55, no. 11, pp. 1335–1343, Nov. 2006.

[4] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "DaemonGuard: OS-assisted selective software-based self-testing for multi-core systems," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst.*, Oct. 2013, pp. 45–51.

[5] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Des. Test Comput.*, vol. 27, no. 3, pp. 4–19, May/Jun. 2010.

[6] F. Reimann, M. Glass, A. Cook, L. Rodríguez Gómez, J. Teich, D. Ull, H. J. Wunderlich, U. Abelein, and P. Engelke, "Advanced diagnosis: SBST and BIST integration in automotive E/E architectures," in *Proc. ACM/EDAC/IEEE Des. Autom. Conf.*, Jun. 2014, pp. 1–6.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Trans. Comput.*, vol. 58, no. 8, pp. 1063–1079, Aug. 2009.

[8] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 1, pp. 88–99, Jan. 2005.

[9] P. Bernardi, M. Grosso, E. Sanchez, and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, Mar. 2011, pp. 1–2.

[10] *Road vehicles – functional safety*, ISO/DIS26262, 2009.

[11] S. Sobek and K. Burke. (2004). PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation [Online]. Available: http://www.freescale.com/files/32bit/doc/app_note/PPCEABI.pdf

[12] ARM. (2008). Application Binary Interface for the ARM Architecture v2.09 available at ARM Information Center web-site: [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html

[13] E. Christopher. (2003). mips eabi documentation available at Cygwin web-site: [Online]. Available: http://www.cygwin.com/ml/binutils/2003-06/msg00436.html

[14] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 1030–1044, Jul. 2011.

[15] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1682–1694, Dec. 2009.

[16] G. Xenoulis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Instruction-based online periodic self-testing of microprocessors with floating-point units," *IEEE Trans. Dependable Secure Comput.*, vol. 6, no. 2, pp. 124–134, Apr.–Jun. 2009.

[17] G. Squillero, "Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs," *Computing*, vol. 93, nos. 2–4, pp. 103–120, Oct. 2011.

[18] F. Corno, E. Sanchez, M. Sonza Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Des. Test Comput.*, vol. 21, no. 2, pp. 102–109, Mar./Apr. 2004.

[19] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, Apr. 2005.

[20] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-SBST methodology for efficient testing of processor cores," *IEEE Des. Test Comput.*, vol. 25, no. 1, pp. 64–75, Jan./Feb. 2008.

[21] M. Scholzel, T. Koal, and H. T. Vierhaus, "Systematic generation of diagnostic software-based self-test routines for processor components," in *Proc. IEEE Eur. Test Symp.*, May 2014, pp. 1–6.

[22] P. Singh, D. L. Landis, and V. Narayanan, "Test generation for precise interrupts on out-of-order microprocessors," in *Proc. IEEE Int. Workshop Microprocessor Test Verification*, Dec. 2009, pp. 79–82.

[23] E. Sanchez and M. Sonza Reorda, "On the functional test of branch prediction units," *IEEE Trans. Very Large Scale Integration Syst.*, vol.23, no. 9, pp. 1675–1688, Sep. 2015.

[24] D. Sabena, M. Sonza Reorda, and L. Sterpone, "A new SBST algorithm for testing the register file of VLIW processors," in *Proc. Des. Autom. Test Eur Conf. Exhib.*, Mar. 2012, pp. 412–417.

[25] P. Bernardi, L. Ciganda, M. De Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line software-based self-test of the address calculation unit in RISC processors," in *Proc. IEEE Eur. Test Symp.*, May 2012, pp. 1–6.

[26] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. Sonza Reorda, M. Grosso, and O. Ballan, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *Proc. IEEE Int. Workshop Microprocessor Test Verification*, Dec. 2013, pp. 52–57.

[27] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. Sonza Reorda, S. de Luca, R. Meregalli, and A. Sansonetti, "On the in-field functional testing of decode units in pipelined RISC processors," in *Proc. IEEE Int. Symp. Defect Fault Tolerance Nanotechnol. Syst.*, Oct. 2014, pp. 299–304.

[28] P. Bernardi, C. Bovi, R. Cantoro, S. De Luca, R. Meregalli, D. Piumatti, E. Sanchez, and A. Sansonetti, "Software-based self-test techniques of computational modules in dual issue embedded processors," in *Proc. IEEE Eur. Test Symp.*, May 2015, pp. 1–2.

[29] (2014). AUTOSAR web-site: [Online]. Available: http://www.autosar.org/

[30] (2015). Doxygen web-site: [Online]. Available: http://www.stack.nl/∼dimitri/doxygen/

Paolo Bernardi (S'03-M'06) received the MS and PhD degrees in computer science from Politecnico di Torino, Torino, Italy, in 2002 and 2006, respectively. Since 2001, he has been with the Department of Computer Engineering, Politecnico di Torino, where he is currently an associate professor. His interests cover the areas of testing of electronic circuits and systems and the design of fault-tolerant electronic systems. He is a member of the IEEE Computer Society and IEEE.

Riccardo Cantoro received the MSc degree in computer engineering from Politecnico di Torino, Torino, Italy in 2013. Since 2014, he has been working toward the PhD degree in the Department of Computer Engineering, Politecnico di Torino. His main research topic is microprocessor testing. He is a student member of the IEEE.

Sergio De Luca is a team leader, project leader, and Functional Safety expert at STMicroelectronics; embedded software development for automotive system-on-chip and real-time applications.

Ernesto Sánchez received the degree in electronic engineering from Universidad Javeriana, Bogota, Colombia, in 2000. In 2006, he received the PhD degree in computer engineering from the Politecnico di Torino, where he is currently an associate professor with the Dipartimento di Automatica e Informatica. His main research interests include microprocessor testing and evolutionary computation. He is a senior member of the IEEE.

Alessandro Sansonetti graduated in computer science from the University of Milan. He manages ST's Automotive Product Group software design teams based in Italy (Agrate B.za, Naples and Catania) and in France (Le Mans). He has participated as an active member of the AUTOSAR consortium as SPI document owner. He has been with ST since 1996.