An efficient strategy for the development of software test libraries for an automotive microcontroller family

An Efficient Strategy for the Development of Software Test Libraries for an Automotive Microcontroller Family

D. Piumatti[1], *Member, IEEE*, E. Sanchez[1], *Senior Member, IEEE,*
P. Bernardi[1], *Member, IEEE*, R. Martorana[2], M.A. Pernice[2]

[1] Dipartimento di Automatica e Informatica (DAUIN), Politecnico di Torino, Torino, Italy
{davide.piumatti; ernesto.sanchez; paolo.bernirdi}@polito.it

[2] STMicroelectronics, Catania, Italy
{rosario.martorana; mose.pernice}@st.com

***Abstract* — With the introduction of the ISO26262 standard in the automotive field, numerous solutions for the in-field and on-line testing have been proposed. Among the several test solutions available, the Built-In Self-Test (BIST) approach is the most used for manufacturing test of chips, while the Software-Based Self-Test (SBST) approach is the most commonly used for on-line test the modern processors. This paper faces a very concrete problem concerning SBST development. In order to address more market demands, semiconductor industries are usually developing families of microcontroller, usually based on similar processors, instead of a single instance. This variety of architectures makes the development of SBST programs a repetitive, time and human consuming activity.**

The main aim of this work is to propose a methodology according with the SBST paradigm that permits to develop test programs able to achieve high coverage on different microcontrollers of the same family. The developed test programs are not showing any significant drop in coverage performance when they are used on different processors included in product of the same microcontroller family. The approach is based on the analysis of the processor hierarchy to identify the common units between the processors of the same family, first of all looking at those that show design differences. The module classification permits than to plan the most effective SBST development.

A segment of industrial microcontrollers developed by STMicroelectronics for the automotive field, adapting many processors belonging to the same processor family, is used as a case of study. The experimental results demonstrate the effectiveness of the proposed approach, i.e., to reach the same fault coverage figures over many processors while dramatically reducing the development time.

***Index Terms* — Automotive microcontroller ISO26262 test, In-Field Self-Test, On-Line Self-Test, Reliability and Testing**

Corresponding author: Davide Piumatti
davide.piumatti@polito.it

I. Introduction

Today vehicles are equipped with very complex functionalities that use of many electronic components. In fact, in a modern vehicle, it is possible to find many different Electronic Control Units (ECUs) placed inside of the vehicle; Actually, in a vehicle's engine, there may exist more than 7.000 semiconductor components able to perform very different tasks. Typical tasks are devoted, for example, to manage safety systems such as the ABS or the ESP, to perform powertrain functionalities, or to improve the end-user experience by means of new applications circumscribed as infotainment [1], [2], [3].

The automotive sector is one of the most dynamic ones, since potential users are always asking for additional but safe and secure features. Actually, the manufacturers of electronic devices for automotive, and in particular the microcontrollers manufacturers, try to launch every few months a new product; all these new products belong to the same family of processors and share some features present in each device. The differences between the family products are mainly focused on the memory sizes, the type and quantity of peripherals available, and the safety mechanisms included in the device. Considering the processors that belong to a given family, in most cases only small variations are applied, e.g., including small modifications to the Instruction Set Architecture (ISA). In this way, the manufacturers exploit the selected architecture and guarantee an appropriate level of compatibility among the products on the same family.

Guaranteeing the correct behavior of the electronic devices composing a vehicle is very complex, and its expected behavior has to be assured in very harsh environments. In fact, vehicles are prone to vibration, noise, extreme temperatures and electromagnetic fields that may affect and degrade the electronic components. The effects of possible faults may lead to significant damages, either from an economic point of view or in terms of consequences for the human users that may produce even human casualties. In safety-critical applications, such as the automotive ones, a set of very good practices have been introduced, trying to guarantee the correct functioning of the electronic devices during their normal life operation. During the last years, the trend is to resort to self-test procedures that operate in-field in autonomous mode. These in-field procedures

have been ruled by the introduction of some safety standards, such as the *ISO 26262* in the automotive field, or ARP-4761 for avionics.

In order to guarantee the system reliability, hardware and software-based approaches have been proposed, e.g., [4] and [5]. In the first case, even though the introduced hardware reaches to assure very high reliability levels, the area overhead and the difficulties to use them without destroying the system status create some difficulties in its adoption. On the other hand, software-based approaches usually reach lower reliability levels than the obtained by the hardware-based counterparts but require very few overheads in terms of hardware and memory. However, the most commonly used implementation methodologies to develop software-based solutions are mostly based on manual processes that involve very long development times making these approaches less attractive for the car manufacturers.

In a family of microcontrollers, the implementation of safety mechanisms based on hardware or software solutions must be implemented and inherited starting from the initial devices to the following ones. These mechanisms should guarantee very good reliability levels for all the components of the whole family. Inheriting hardware-based solutions is usually handled by the use of commercial tools that implement these mechanisms almost automatically. On the other side, software-based solutions are rarely inheriting due to the implementation methodologies that in most cases tackle only one processor at a time.

In this paper, we present a development methodology for software-based solutions oriented to provide a quick and cheap strategy that considers a whole family of microcontrollers at a time, instead of addressing each single processor cores separately. We propose to exploit the similarities among the different components of a family of microcontrollers during the development process. In particular, we define a *portable classification* topology that permits us to take advantage of the processor similarities during the early development stages of the SBST programs. Additionally, we define how to develop a set of test programs for the most common modules available in a family of microcontrollers in order to reduce the development time of the software test libraries.

Through a very consistent set of experiments, run in a family of automotive oriented devices manufactured by STMicroelectronics, we experimentally observed that the generation of a software-based solution for a family may require the same development times as of the individual approach for each single products of the portfolio. The family oriented approach is also guaranteeing very high fault coverage (FC) levels for all components of the considered family. In this direction, to avoid the loss of effectiveness of the test programs from one processor to another, a threshold is defined. The identification of the minimum FC threshold is discussed in this paper. Additionally, the proposed approach describes how to maximize the processor FC by reusing test programs previously developed.

The paper is organized as follows: Section II proposes a broad background concerning the on-line self-test, with a particular emphasis on the Software-Based Self-Test (SBST) approach. The development steps of a new processor family are also discussed in Section II. In Section III the proposed approach is discussed and analyzed; it allows the development of software test programs that are easily portable between the different processors of the same family under examination. Section IV reports the case study, i.e. the SPC58 processor family used in this work is analyzed. Section V reports the experimental results that support the proposed approach. Some industrial problems related to the development of the test programs are reported in Section VI. Finally, Section VII close this work with some conclusions.

II. Background

This section proposes an overview of the safety standards used in different fields, with a particular emphasis on the automotive field. The motivations related to the in-field self-test are introduced and the two main categories of safety mechanisms used to perform in-field testing are reported. Later, an overview of the software-based test approach is provided. The structure of a generic Software Test Library (STL) used for testing a processor is discussed and analyzed. Finally, a description of the design process of a new industrial processor is shown.

A. Safety standards

The IEC 61508 [6] is an international standard introduced at the end of the 1990s. The International Electrotechnical Commission has proposed this standard with the goal of introducing some methods to apply, design, deploy and maintain automatic protection systems called safety-related systems. As the standard reported, it defines the functional safety as: “part of the overall safety relating to the EUC (Equipment Under Control) and the EUC control system which depends on the correct functioning of the Electrical/Electronic/Programmable Electronic Safety-related systems (E/E/PES), and other technology safety-related systems and external risk reduction facilities.” [6]

The fundamental concept described in the standard is that any critical-system must work correctly or fail in a predictable (safe) way. As a consequence of the *IEC 61508*, many specific standards are introduced for different application fields. In the medical field, the *IEC 62304* standard has been introduced, the *EN 5012x* has been introduced in the railway field. The *DO-178* standard is used in the aviation field, while in the automotive industry the *ISO 26262* standard has been introduced.

The *ISO 26262* [7] is an international standard introduced in 2011. The target of this standard is to define a functional safety metric for all Electrical/Electronic Systems used in automotive applications. The *ISO 26262*, in opposition to previous standards, introduced the concept of controllability [4]. The controllability is the ability to avoid a hazardous event by an action taken by a driver or by a system. The standard introduces four Automotive Safety Integration Level (ASIL) classes depending on severity, probably of exposure and controllability of dangerous events. The standard faces the steps to analyze the hardware failures of the electrical and electronic parts of cars. In addition, the *ISO 26262* classifies the faults in some categories [4]:

- Perceived: This fault is perceived by the driver, but the fault is not detected by a safety mechanism in a prescribed time.
- Detected: This fault is detected by safety mechanisms in a prescribed time.
- Latent: This fault is neither detected by a safety mechanism and it is not perceived by the driver.
- Safe fault: Fault whose occurrence will not significantly increase the probability of violation of a safety goal.
- Residual fault: The effect of this fault does not affect the system.

B. Safety mechanisms

Different solutions facing the problem of the in-field self-test are proposed to be compliant with the introduced safety standards. Roughly speaking, the proposed solutions can be categorized into two main categories that separate pure hardware and software-based solutions; however, in the last years, hybrid proposals are also finding some space as safety mechanisms. Typically, the safety mechanisms are targeted to detect the permanent stuck-at faults, but it is possible to extend the test strategies to other fault models as the delay faults, the transient faults or the bridge faults. In every case, the FC figure can be evaluated with a fault-simulator software tool [8] [9]. In the automotive field, the FC is also called Diagnostic Coverage (DC) [7].

1) Hardware-based approaches

Belonging to hardware-based approaches [10], the Logic-BIST (L-BIST) is one of the most popular approaches. In this approach, a state machine applies some non-functional test patterns to the Unit Under Test (UUT). The BIST system collecting and checking the results at the end of the process. When applied to a processor core, this test strategy can be used only during the power-on because it requires the system to be in a specific test-mode. On the other side, some safety mechanisms can be based on duplication and triplication of the UUT. In particular, the Triple Modular Redundancy (TMR) [11] technique uses three implementations of the same UUT and the output signals of these modules vote adopting a voting mechanism. The most basic voting algorithm is the majority voter, where the voter selects the most common output. The Lockstep configuration is a redundant system based on UUT duplication. Regarding the memory testing, two approaches are most used today: the Memory-BIST (M-BIST), and the Error Correction Code (ECC) approach. The first one uses a March test sequence to test the RAM or Flash memory [12]. The M-BIST approaches write and read sequence of test patterns in the memory cells oriented to detect different types of faults; these approaches modify the content of the memory. For this reason, the M-BIST can be used only at the power-on of the processor when the memory contains useless values. On the other hand, the ECC one uses a redundant code to perform an on-line check [13]. All hardware-based approaches require the instantiation of additional hardware to perform the testing processes. The amount of added hardware to perform the test can be a significant part of the whole device area.

2) Software-based approach

In order to use this approach, a test library able to detect the possible permanent hardware faults must be developed. The Software-Test Library (STL) is a collection of software programs able to excite the possible faults inside of the processor and the peripherals surrounding it. This strategy, initially proposed in [14], has been studied by different research groups as described in [15] [16] [17], and later extended targeting the automotive field [5]. Currently, the STL approaches are used by different companies to mainly test their own processors, for example: STMicroelectronics [18], Infineon [19], Cypress [20], Renesas [21], Microchip [22] and ARM [23]. The STL technique is based on the so-called Software-Based Self-Test (SBST) paradigms. The SBST consists of letting the CPU running a sequence of instructions to excite and propagate the faults that may affect the digital circuit [24]. The processor is periodically forced to execute the self-test code [5] able to detect the possible occurrence of permanent faults in the processor core itself, or in the peripherals connected to it. Such procedures are developed to activate possible faults and report their presence.

Usually, the test programs return a value called signature. In the presence of a fault, the signature value produced by the test is different from the expected one. The signature value is produced by accumulating the results of the assembly instructions that perform the test. It is a good practice to develop a test program for specifically testing one unit of the processor. As described in [25], a test program can be developed mainly resorting to three different approaches: *ATPG-based* approach, *deterministic* approach, and *evolutionary-based* approach. The first one uses the test patterns generated by an ATPG tool [26] to test a functional unit of the processor. The *ATPG-based* approach is very powerful to test the arithmetic and logic units as the adder, the multiplier, and the divider units. The test program executes an assembly instruction able to apply the ATPG functional test pattern; for example, using the test patterns as the operands of an *add* instruction to test the adder unit. The results of the test instruction are used to produce the signature of the test.

In the *deterministic* approach, the test program is developed studying the Unit Under Test (UUT). A deep knowledge of the UUT is necessary to develop a good test program and for implementing a specific test algorithm. Some examples of deterministic test algorithms are available in the literature, for example, in [27] for the Register File Unit and in [28] for the BTB unit. A testing algorithm to test the decoder unit is proposed in [29], while the test for the FPU is proposed in [30].

The last possible approach to develop a test program is the *evolutionary-based* approach. A first pseudo-random set of test programs is written, then, resorting to some genetic operators. The test programs are modified generating new and hopefully better test programs. The goodness of every individual or test program is evaluated against a given metric, e.g., the fault coverage reached by the program in the UUT. Then, the best individuals are selected for generating the next set of individuals. The evolutionary algorithm is executed until a stable condition is reached. As an example, an evolutionary optimizer called μGP [31] has been used to evolve the test programs.

The test programs that belong to a STL can be classified according to their ability to be integrated with the mission software application. The STL is usually composed of *intrusive* and *non-intrusive* test programs [32]. The former ones influence the behavior of the operating system or the mission software application because the intrusive test needs to take the total control of the processor to perform the test; e.g., test programs triggering exceptions, or using very special addresses in the RAM memory, or manipulating special registers. The *intrusive* tests need to be executed at the power-on or power-off, for example before launching the Operating System. On the other hand, the *non-intrusive* tests can be executed by the Operating System as a simple application because they do not require special conditions. The *non-intrusive* tests are usually performed with the processor configured in user-mode and they are periodically executed at run-time scheduled by the Operating System.

A last test program category, which is also usually included in a STL is the so-called Instructions Self-Test (IST). The goal of the IST test programs is to execute at least once all the assembly instructions of the Instruction Set Architecture (ISA) supported by the processor.

Compared to the hardware-based solutions, the STL presents many advantages, such as the ability to perform the test at the boot time as well as at run-time; the test programs are executed at-speed (i.e., at the circuit nominal frequency); and the STL does not require any hardware modification. On the other hand, the STL programs require to be allocated in the flash memory, and according to the execution schedule, these programs require be executed concurrently with the mission application. In the automotive sector, for example, the memory occupation as well as the execution time must comply with the system constraints in order to do not impair the execution of the actual application. In an automotive solution, the full flash memory occupation counts with about 200KB, while a single non-intrusive test program must spend at most 255 clock cycles [32], at every run. The STL-based approaches still present a serious limitation due to the difficulty of both writing efficient and effective test programs and devising suitable methodologies for test application.

In order to allow the STL test programs to be compliant with the mission software environment, a viable solution is the adoption of the Embedded-Application Binary Interface (EABI) [33]. The EABI specifies standard conventions for the data types, the registers usage, the stack frame organization, and the function parameter passing of a software program. Thus, every test program includes an EABI prologue and epilogue, able to save and restore the mission status.

In order to assess the test program suitability, the test programs are evaluated through a fault-simulation process as described in [34] and [35]. Each test program is evaluated targeting only the faults in the UUT as described in [5]. When the Fault Coverage of the single units reach a good level, a synchronization process [5] is performed, i.e., all test programs are fault-simulated targeting the whole processor obtaining a general Fault Coverage of the processor. Following this process, the fault-simulation time is reduced as shown in [5], and it is possible to take advantage of the cascade phenomenon [5][36]. The cascade phenomenon consists of exploiting the beneficial effects on the fault coverage introduced by a test program, devised for a specific unit, on the other units of the processor.

3) Hybrid approaches

Among the test strategies proposed by the scientific and industrial community, a new third category is currently under development: the Hybrid approach, see [37], [38], [39], [40], [41], [42]. The hybrid approach merges the software approach with the hardware one. The hybrid approach tries to take advantage of the positive features of both techniques. It is able to reach a high fault coverage, as the hardware-based approach, with the ability to work on-line, as a software-based approach. The idea is, for example, to use a hardware test architecture driven by a software test program for applying some test patterns. The architecture works in a similar manner as the L-BIST approach, but it is not limited to work at power-on. The hybrid approach allows to perform periodical on-line self-tests. In order to integrate these techniques in new devices, it is required to modify the hardware device; for this reason, hybrid approaches are not targeted in this paper.

C. STL architecture

The goal of this subsection is to show how an STL works, considering the final user point of view. Usually, two different sets of Application Programming Interfaces (APIs) are available, one for the tests performed at boot-time and one for the run-time test programs.

At the boot-time, a single API calls a software task able to execute the test programs, as shown in Figure 1. Typically, the *STL_BOOT* is performed at the start-up before loading the Operating System. A *Test_init* function prepares the processor to perform all the test programs. In particular, the *Test_init* configures the interrupt controller to manage the interrupt requests generated by the test programs, initializes the RAM memory, disables all peripherals and configures a watchdog timer to avoid the program to be stuck in an infinite loop. The *Test_loader* function launches each test program and checks the signature value against the expected one. In case of a test program fails the *Safe_state* function is performed. The *Safe_state* function freezes the ECU in a safe state. If all the test programs return the expected signature, a *Test_deinit* function restores the processor state. Finally, the Operating System is launched.

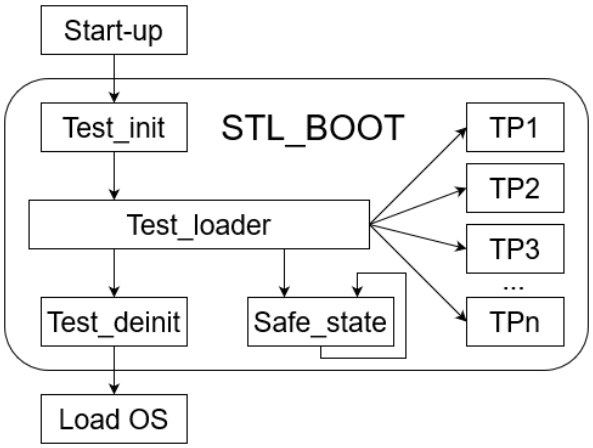

Figure 1: STL architecture for the boot-time tests

The non-intrusive test programs are performed at run-time, in this case, APIs complaints with the AUTOSAR standard is required [43]. Three APIs are usually implemented: *Prepare_STL*, *Call_test_routine* and *Return_last_test_state*. The *Prepare_STL* function initializes the variables used by the loader of the test programs. The *Call_test_routine* executes the test whose ID is passed as a parameter. While the *Return_last_test_state* indicates if the last test program executed has detected a fault or not. In the presence of a fault, the system must be placed in a safe state.

D. Design process and industrial production timeline

The development of a processor is a long and complex process that requires many steps to reach the final product [44]. Analysing the process at a high level, the features of the new device are initially defined and described. This higher level is known as the *behavioural* level. In the case of the processors, the general architecture and the Instruction Set Architecture (ISA) is established. In the next step, a formal description of the processor using a high-level language is performed. In this step, called *Register Transfer Level* (RTL), every single unit of the processor is described and its behaviour verified. The RTL is independent of the technology that will be used to implement the processor, but the RTL description is sufficiently detailed to allow the synthesis of the digital circuits. Furthermore, the RTL description allows us to perform formal verification of the processor using a logic simulator [45]. The next step is automatically generated from the RTL level through a synthesis process, this new level is called *Gate-Level*. At *Gate-Level* the circuit is described at the level of logical gates considering a specific technological library. In the last step called *Switch-Level*, the processor is described at the transistor level. The *Switch-Level* is used to generate the chip layout. The layout is the description of the geometric information necessary to activate the final production process. An integrated circuit consists of a succession of silicon, oxide and aluminium layers that must be arranged in a certain way to create the transistors and the connections between them. Each of these layers corresponds to one or more production processes that are regulated by one or more masks. The complete set of the masks derived from the layout defines all the operations to be performed in production to create the final *die*. The *die* is the thin plate of semiconductor material on which the electronic circuit of the integrated circuit has been made. Finally, the *die* is closed in the plastic package, and the wires bonding is realized to connect the *die* with the external package contacts. Obtained the first sample of the new processor, a verification phase is performed. The aim of the verification phase is to check the device to an electrical point of view, e.g. the electrical power consumption of the processor.

Typically, the processor manufacturer releases a Microcontroller Abstraction Layer (MCAL) [46] package containing the drivers and APIs for using the processor itself. The MCAL is a software module that directly accesses the on-chip MCU peripheral units mapped in memory. The MCAL contain, for example, a set of drivers for the peripherals as the GPT (General Purpose Timer), the WDG (Watchdog), the MCU (Micro Controller Unit) as the MMU (Memory Management Unit) or the MPU (Memory Protection Unit), and for all communication devices as the CAN bus, the LIN bus, the Flex Ray bus, the Ethernet and the UART interface. In the automotive field, the MCAL structure is defined by the AUTOSAR standards [46]. The MCAL package is developed and tested by the manufacturer of the processor.

The whole development process of a processor used in an embedded system may take about one year from the initial behaviour description to the first physical sample. The verification phase may require an additional 4 months for checking the physical device, and about 7 months to produce and check the MCAL library [47][48].

The technical and commercial planning are very important aspects, among the aspects that concern the development of a new processor. In particular, the roadmap of the new processors is defined in order to establish the development plan and the production plan. The aim of these plans is to establish the characteristics of each processor. In particular, to establish the characteristics that change from processor to processor over time; for example, the size of the memories or the number and type of peripherals present in each processor [49]. In addition, the financial investment plan is established for the future years. The economic plan is associated with each development step and with each production activity [50]. In general, from a processor to another processor of the same family two develop roads are available. In the first one, the "child" processor is built by reducing the features of the "father" processor. In the second one, some features of the "father" processor are redesigned and improved. With these two approaches, it is possible to produce a wide range of processors belonging to the same family. However, the basic structure of the processors remains unchanged for all processors of the same family. The family tree of the processors can be produced considering the two possible develop roads.

III. PROPOSED APPROACH

This section discusses the proposed approach, the aim is to reduce STL development time for each processor of the same family; Secondly, the proposed approach allows to identify a structure for the development of test programs. The test programs developed with the proposed approach are efficient on different processors, i.e., the ability of the test programs to detect faults does not degrade from one processor to another processor of the same family.

The proposed approach is supported by three different items of the new processors family: 1) the family tree of the new processors; 2) the development plan of the processors; 3) the features of each processor of the new family. All these three items, discussed in Section II, must be available before starting with the development planning of the portable tests.

Briefly, the proposed approach is based on the classification of the processor's units. The proposed unit classification, called *portable classification*, is used to define how to develop the test programs for each unit.

In the following, the first subsection introduces the idea of portable test programs; the second subsection reports the proposed portable classification, and finally, the last subsection discusses the proposed porting approach.

A. Definition of portable test program

A portable test program mainly operates in a functional way, i.e., it works independently of the hardware implementation of the unit that it tests. This consideration is useful in order to abstract the test program from the hardware, and in particular from the specific UUT.

For example, the *ATPG-based* approach should be avoided when aiming to produce portable test programs. In fact, the *ATPG-based* approach tries to find a set of optimal test patterns for a given hardware implementation of the UUT. These test patterns are generated by special ATPG tools that operate at the gate-level. Clearly, a new synthesis of the circuit using different technological library or different synthesis parameters produces a different gate-level implementation, that requires a new set of test patterns. Thus, the ATPG test patterns generated for the UUT of a processor are not suitable for testing the same unit in a new processor, since a consistent loss of FC is predictable.

In a similar way, the evolutionary-based approach should be also avoided aiming to developed portable test programs. Actually, in the evolutionary approach, the test program evolves according to the specific gate-level network in order to obtain high FC values. The test program developed is therefore specific for a single implementation of the UUT, and also in this case, there is a considerable decrease of the FC by reusing the test programs developed with the evolutionary-based approaches.

In general, all the test programs developed using approaches based on a direct exploitation of the gate-level information are not suitable for the development of portable test programs. This is due to the considerations of the synthesis phase above described.

As a matter of facts, in order to develop portable test programs, the *deterministic* approach is therefore preferred because it is based on the functional study of the UUT at RTL. Obviously, the *deterministic* approach lengthens the development time of the test programs because a study of the UUT is required, as described in subsection B.2 of the Background. However, this development phase is performed only one time on a processor of the new family. In the long term, there is a considerable saving of time and resources necessary to develop the test programs.

B. The proposed portable classification

This classification analyses the units of a processor with respect to similar versions present in other processors of the same family. Four possible categories are analyzed and discussed:

The *EXCLUSIVE* unit: The *exclusive* unit is present only in the processor under examination, and it is not present in other processors of the same family. In general, the *exclusive* units are included in a processor to optimized some specific operations requested by the customer.

The *SHARED* unit: In contrast to the previous category, the *shared* units are present unchanged in many processors of the same family. In a more general sense, it is possible to consider these units as belonging to the processor family.

The *REDUCED* unit: These units are included in many processors of the same family, but from processor to processor these units miss some functionalities. For example, the multiplier unit able to perform operations on 64-bit operands has been simplified, and in its next version the multiplier performs operations only on 32-bit operands.

The *INCREASED* unit: Similar to the *reduced* unit, the *increased* units are present in many processors of the family. Furthermore, the *increased* units improve their functionalities in the next version. A possible example is the extension of the instruction set of the processor. With the addition of some instructions, new features must be implemented in the processor units.

The *REDUCED* and the *INCREASED* categories can be considered as a sub-category of the *SHARED* one.

C. Porting methodology

In this section, the proposed approach to develop a portable STL is shown and discussed. The steps of the proposed approach are shown in Figure 2.

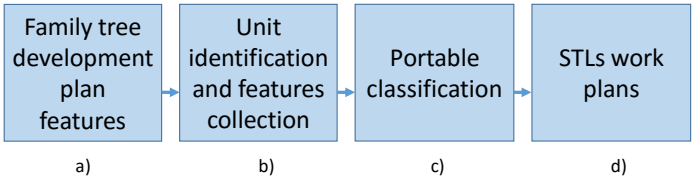

Figure 2: The classification of the units of the processor family

- *Step a*: The family tree of the processors family, the release times to the market of every device, and the features of each processor are gathered. At this point, it is important to determine when every phase of the development plan for every processor core in the family will take place. In particular, it must be defined the delivery times for the RTL, GATE, and Software MCAL. Figure 3 shows an example of a family tree composed of 5 different processors. From the figure, it is possible to see that the processor 1 must be delivered at M15, at the end of the fifth quarter (Q5), the RTL of the processor 1 is to be released during Q0 and the gate level near the end of Q3. It is possible to notice also that there is a dependence between the RTL of the processor 1 and the ones of processors 2 and 4.

- *Step b*: The units of the different processors of the family are identified and the features of every unit extracted. It is important to identify the features in every processor and their evolution on the other cores in the whole family. Additionally, resorting to [5], a first analysis of the processor units is performed and the different units are classified according to their functionality. Five categories are identified: the first one includes all the functional units. These units execute specific operations in the processor such as the addition, shift, division, and the logic operations, these units are labelled as *FUNCT* units. The second category is named *SPECIAL*. The units belonging to this category are associated with the management of the instruction flow or the memories; the exception unit, the memory management unit and the branch predictor belong to the *SPECIAL* category. The third category only includes the processor *REGISTER FILE*. All the General-Purpose Registers and the Special-Purpose Registers belong to the *REGISTER FILE* category. The program counter unit and the effective address calculation unit belong to the *ADDRESS* category. The last category is the one that

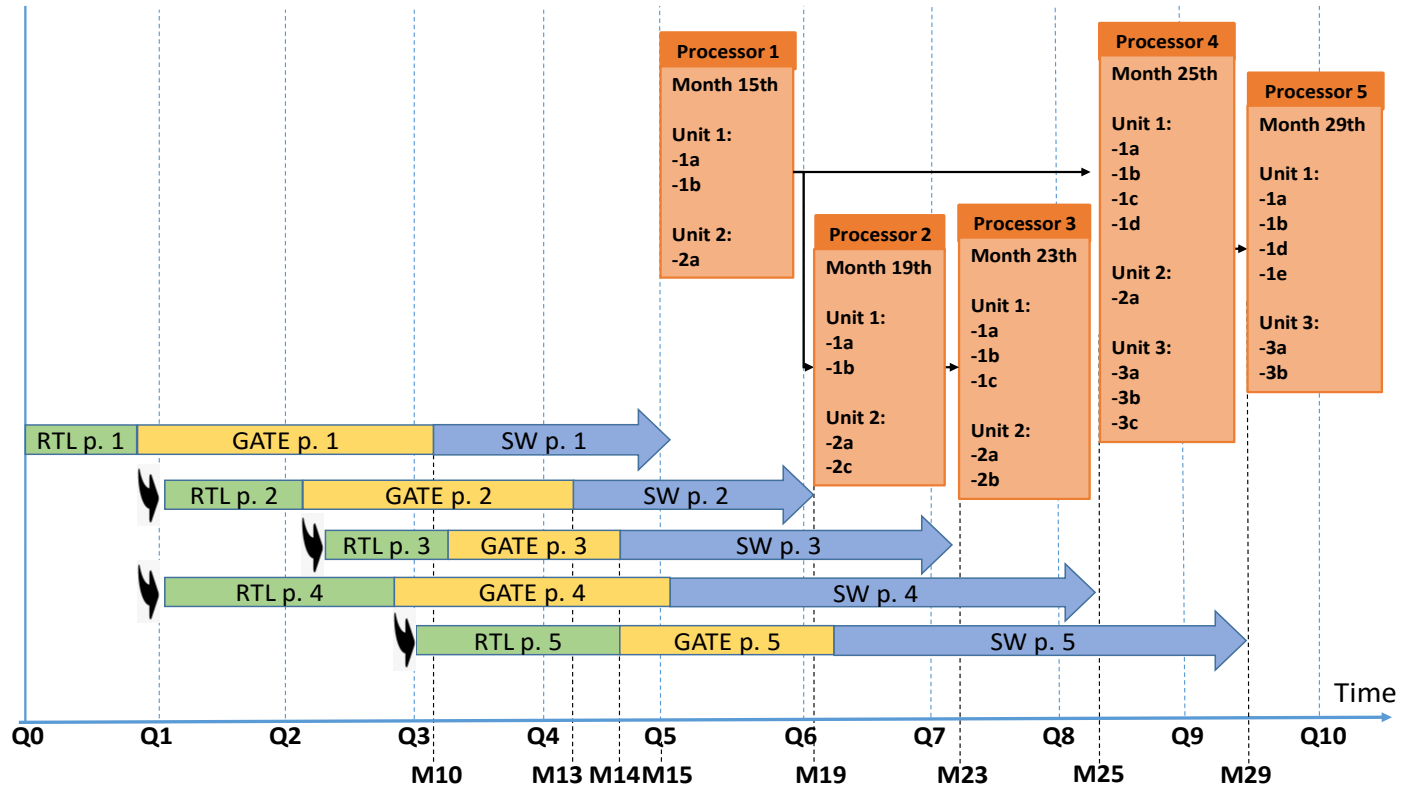

Figure 3: Family tree example

includes *CONTROL* units. All the units able to management the pipelines belong to the *CONTROL* category.

According to [5] the test program development process must follow an almost sequential order, starting with the *FUNCT* units, followed by the *SPECIAL*, *REGISTER FILE*, *ADDRESS* and *CONTROL* ones. In some intermediate points, a general synchronization is performed in order to take advantage of the beneficial results of the developed programs in the whole processor fault coverage. The development strategy proposed in [5] does not consider the portability of the programs, actually, it uses extensively non-portable solutions. Figure 4 shows a possible development process based on this technique while considering the delivery times for the family processors in the previous figure.

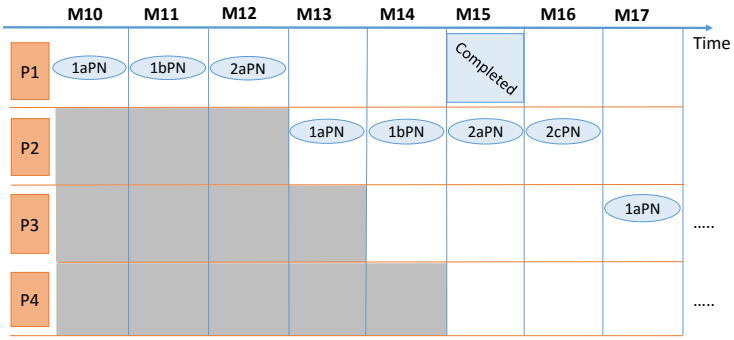

Figure 4: Development process in [5]

In this case, it is possible that the development process takes so long time and the expected delivery time of the processor as well as the STLs may not coincide.

- *Step c*: The processor units are classified according to the portable classification (see Section III.A). For the units classified as *shared*, *reduced* or *increased* the test programs must include portable test programs for every feature of the unit. On the other side, units classified as *exclusive* require a traditional non-portable test program. In general, portable test programs are structured in a modular way considering the unit features, i.e., an independent sub-test is developed for each feature. Once the portable sub-tests have been developed, an additional non-portable sub-test is usually required. This new non-portable test guarantees the targeted FC, i.e., the aim of the additional sub-test is to cover the gap of remaining FC, with respect to the expected FC value. In the example reported in Figure 5, the processors units are graphically classified as *shared* (blue borders) and *exclusive* (green borders). Additionally, the figure also highlights when a new feature appears in the development process for the shared units; this is named *shared new*, and is indicated by a blue arrow.
- *Step d*: The *STL Development Plan* (SDP) is produced in this phase, is then important to define the number of *Development Units* (DU) composing the project. A DU is a team composed by at least one test engineer and an appropriate computation system where the development process is computed; here we assume that initially there is a unique DU in the development process, and that a DU is able to develop only a portable and a non-portable test program

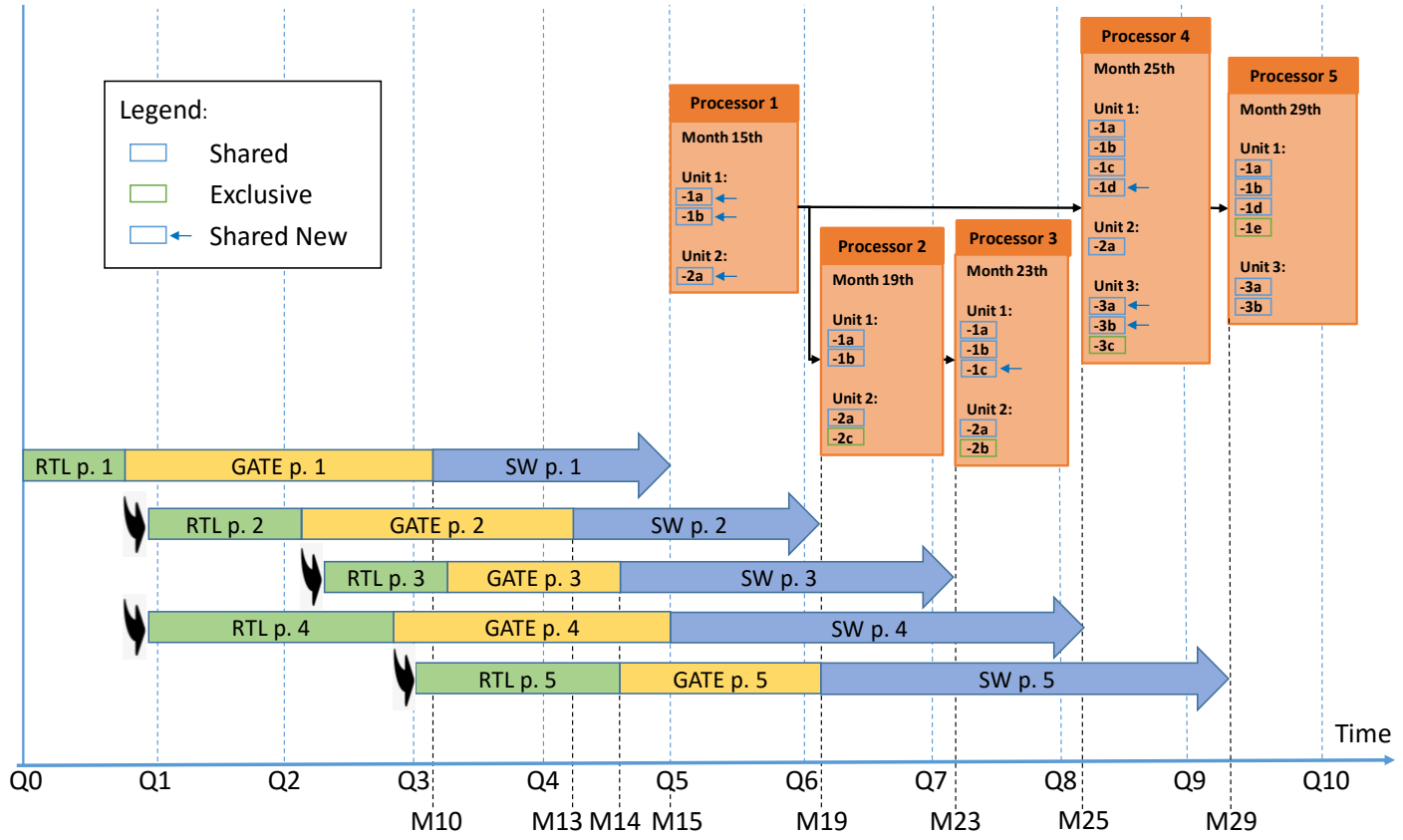

Figure 5: Portable classification of the processors family

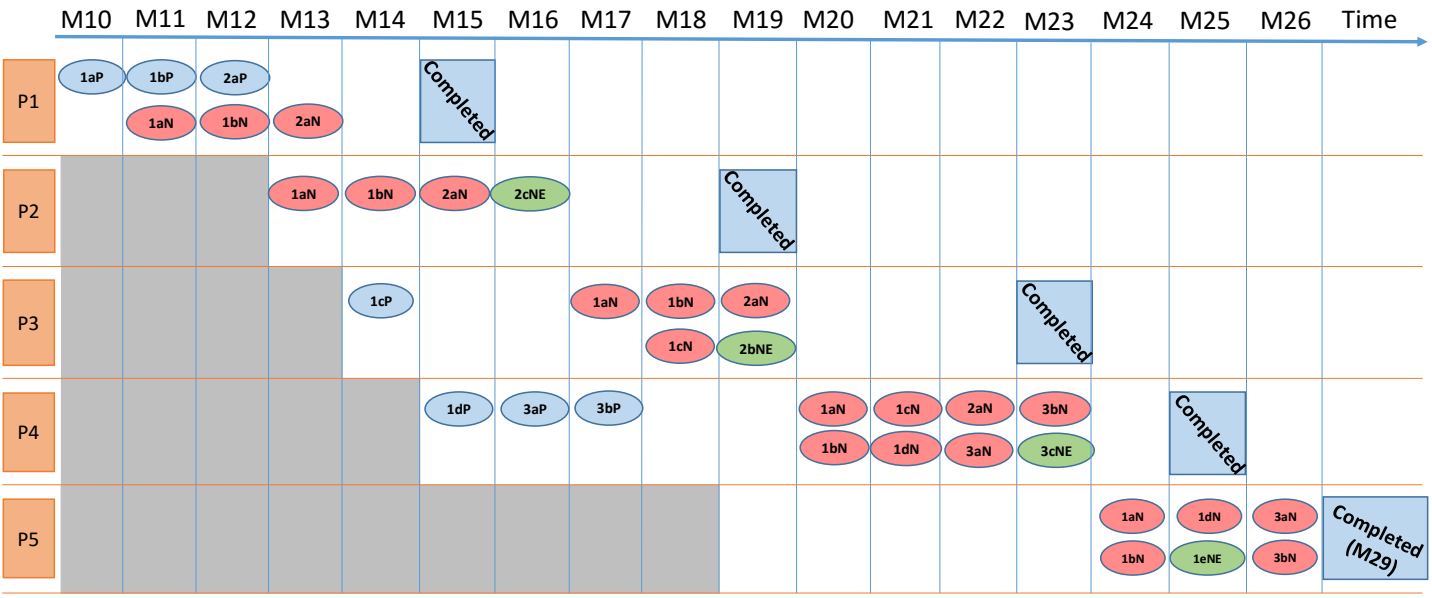

Figure 6: SDP for a unique DU

every month, or even two non-portable test programs but never two portable test programs.

The development plan creation is divided in two phases: in the first one, an initial SDP is proposed ordering the development of the tests with respect to the release date of the processors to the market and the classification reported in *step b*, as the one reported in Figure 4. In this figure, the test development of the portable and non-portable parts for every unit are represented together; for example, for the processor 1 (*P1*), the development of the test programs for the feature *a*, of the unit *2*, is performed during *M12* for both parts: the portable *P* and non-portable *N*.

At this point, the second phase starts incorporating the information about the portable classification made in *step c*.

For every unit, the portable part must be developed before the non-portable part.
Thus, the following algorithm is applied to the original development order:

1. Allocation of the portable part of the shared new units considering the processor delivering time. In this allocation, given 2 portable tests to develop, the priority is given to the one with the nearest delivering time
2. Allocation of the non-portable units in the remaining free slots considering:
   a. The portable part should be already developed;
   b. Given 2 non-portable test to develop, the priority is given to the one with the nearest delivering time

Applying this algorithm to the example in Figure 4, let us with the development plan provided in Figure 6. In the figure, the portable and non-portable parts are represented using blue and red elements, respectively; in addition, the green elements represent the development of non-portable test programs for exclusive features that need not share the results later to other processors.

The development must consider the case in which the number of DU increase, for example, it may happen that at the very early development steps, the number of DU is only one, but after a while, when a parallel process is necessary, the number of DU is increased to two or even more resources if available. The same example described previously is developed considering additional DU in Figure 7. During the definition of the development plan, the order of development of the individual tests for the individual features of each unit is modified to anticipate the development of the units classified as *shared* according to the previous algorithm. It is expected that the resulting test programs are then ported to the other processors of the same family. Porting a test program to processors containing *reduced* units need minor efforts since the test program is a reduced set of the original one. In contrast, for the units classified as *increased*, the test programs need additional test programs that are specific for testing only the new functionalities. At the end of the generation process, every portable test program is combined with a non-portable test program.

The purpose of this second non-portable test program is to compensate for small FC drop on the considered unit. In order to accomplish with the marketing times offered by the microcontroller producers, it is very important to synchronize the development of the STL with the final production steps of any microcontroller in the family.
For the sake of simplicity, it is assumed that any development process needs the same time unit ΔT, in this case represented by 1 month, and as reported in the Figure 7 for example, in M13 and M14 a second and third DU are introduced to the development team. It is possible to notice that in the time M14 one portable test program, two exclusive and three non-portable sub-test programs are developed simultaneously.

During the last step of the proposed algorithm, the test program structure is created for any one of the units belonging to the processor family. Figure 8 shows the structure of the test program for *Unit 1* of the example. The whole test program for the *Unit 1* needs to consider the five different features that the unit may count with in any processor implementation. In Figure 8, the portable sub-tests are shown in blue, while the non-portable sub-tests are shown in red. In Figure 8 are reported in

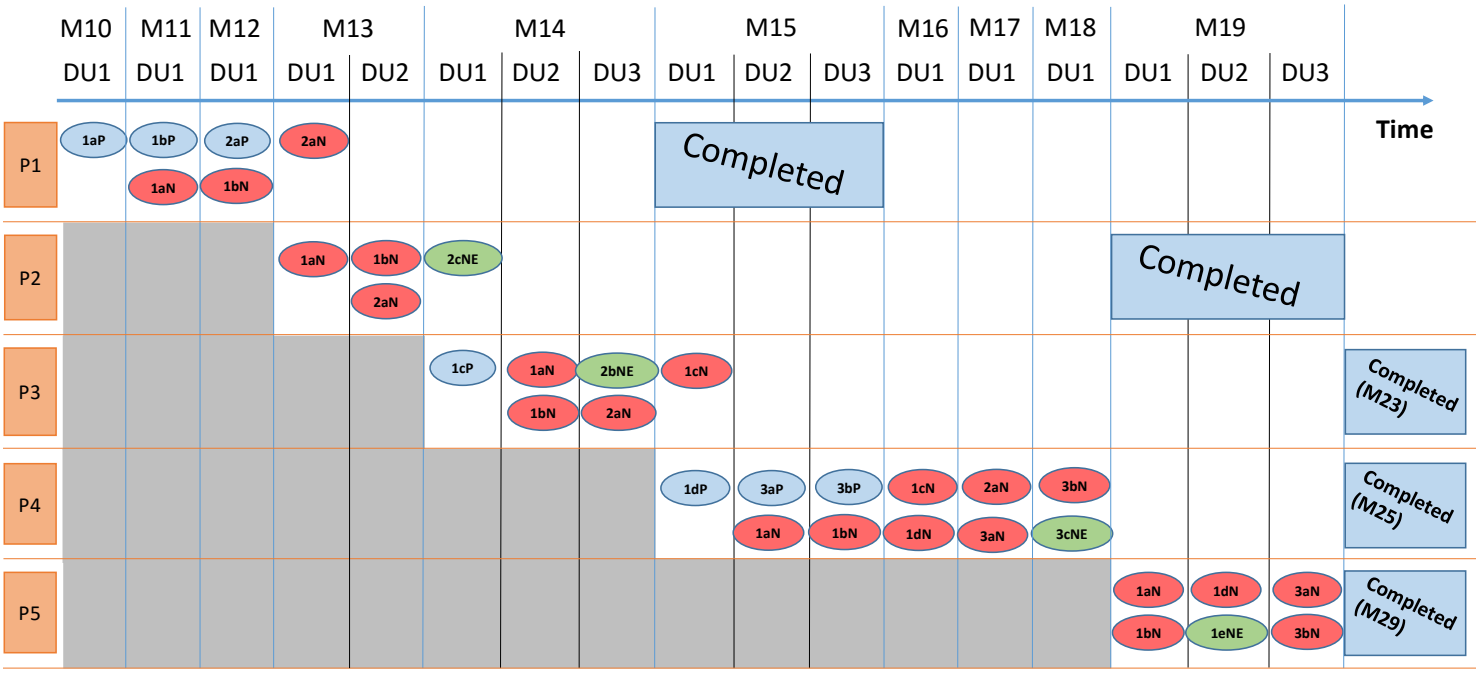

Figure 7: SDP for increasing DUs

green the exclusive sub-test. Initially, the test program is developed for the *Processor 1 (P1)* feature *a* and *b* as discussed previously, and then ported to the *Processor 2 (P2)* and *Processor 3 (P3)*. For any processor, it is necessary to develop a non-portable sub-test independently. In the *Processor 3 (P3)* a new portable sub-test for the feature *c* is developed similarly. The test program for the feature *d* is developed for the *Processor* 4 (*P4*) and ported to *Processor* 5 (*P5*).

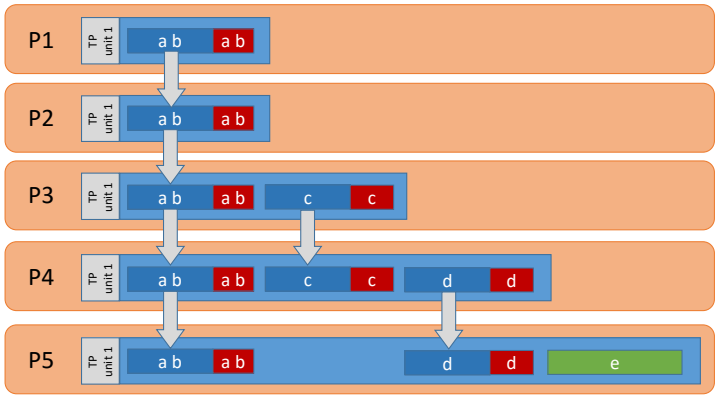

Figure 8: Test program structure for a common unit in the processor family

IV. Case Study

This section introduces the case study. Different STLs have been developed, according to the proposed approach, for testing different automotive processors produced by STMicroelectronics. The STLs of the different processors have been developed with the approach proposed in Section III. Initially, this section provides an overview of the *SPC58* processor family. Afterwards, this section reports the most important features of some of the SPC58 family processors.

A. The SPC58 processor family

The SPC5X processors developed by STMicroelectronics are specifically developed for the automotive sector and for the different applications required by this sector. The last family designed by STMicroelectronics is the SPC58, it is available to the ECU development engineers since 2016. In this new family, there are numerous processors operating in a multicore context. The introduction of multicore architectures allows greater data computing on the ECU, it is necessary to meet the new needs of the automotive market. Today, the vehicles have sophisticated management and control systems for their parts; such as the engine, the suspension and the management of safety systems on board. Moreover, in the new vehicles, there is great importance to the infotainment applications. The aim of the infotainment applications is improving and facilitating the driving experience. Moreover, with the future introduction of the autonomous guide, the necessary computational abilities on the ECU remains a great technological challenge. The SPC58 family processors are based on the 32 bits Power-PC architecture able to work up to 200MHz. They have large flash memory, from 1MB up to 10MB, and different types and sizes of RAM memory. The RAM memory shared between all cores and the private local RAM memory for each core. These different RAM memories are accessible by the cores with different time latency. The processors are equipped with numerous communication channels; such as the CAN bus, the LINFLEX bus, the UART, the USB interface and the Ethernet interface. In the SPC58 architecture the cores, peripherals and memories are divided over several AMBA BUS; to guarantee the best performance in terms of access speed. The Hardware Security Module (HSM) is introduced in the SPC58 family for managing the security aspects of the communication interfaces. The SPC58 family is segmentation in more lines of products [51]. The processor of the *A* and *M* lines are specifically developed for the engine propulsion control and for the transmission control. The *A-line* and *M-line* processors are high performance processor to managing complex real-time control software. The Digital Signal Processing (DSP) features are available on these processors. The processors of the *P* and *L* lines are used to managing the electrical sensors and to elaborate the measured performed by the acquisition systems. They are equipped with different 12 bits Analog-Digital Converter (ADC) able to work at high speed. The processors of the *D*, *B* and *C* lines are thinking to the network and low power applications. The description of the SPC58 family processors considered is shown in Figure 9, it is temporally organized. Figure 9 shows the period in which the processor and the MCAL software package are available to the customer. Moreover, the processor construction technology, the maximum work frequency and the number of the cores present are shown in Figure 9.

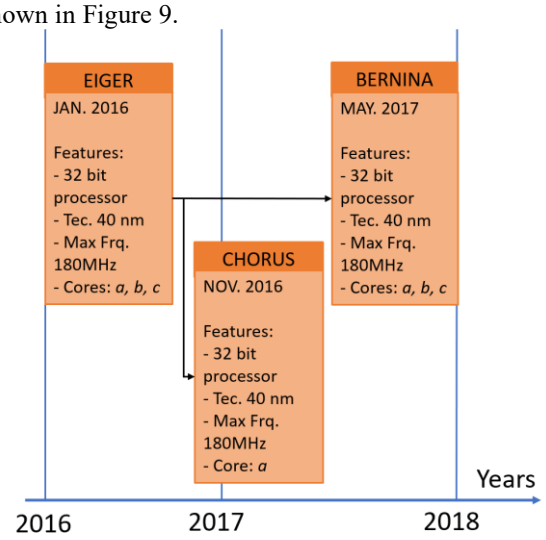

Figure 9: Roadmap of the developed processors of the SPC58 family

B. Compare the processors of the SPC58 family

This subsection provides a description of the cores used in the SPC58 family processors. Moreover, the features of the core units are also compared. In this paper the processors *Eiger*, *Bernina* and *Chorus* are considered.

1) Eiger

The *Eiger* processor (SPC58NEx) [52] is a multicore processor equipped with 6MB of flash memory, 600KB of shared RAM and 64KB of local RAM for each core. The processor is

organized in two AMBA BUSs on which the different peripherals are connected. The processor is equipped with three cores (*core a*, *core b* and *core c*) of different type. All the cores are based on a double issues pipeline with five stages each, and all cores are able to execute the ISA Variable-Length Encoding (VLE) [53] assembly instructions and the Book-E [54] compatible VLE assembly instructions. In addition, the *core a* also supports the Lightweight Signal Processing (LSP) [55] instructions. All the cores are equipped with 32 General-Purpose Register (GPR) at 32-bit each, and some Special-Purpose Registers (SPR) used for configuring the core features. Moreover, all cores are equipped with an Embedded Floating-Point Unit (EFPU2) and a Performance Monitor unit. The Performance Monitor is used for profiling the core activities at run-time, it is composed of 4 counters able to count different events. The Nexus3 debug module is present in both core types. The *core b* is equipped with 8KB of first-level (L1) instruction cache memory and 4KB of first-level (L1) data cache memory; while the *core a* has 8KB of L1 instruction cache. All cores have a Memory Protection Unit (MPU) and don't have the virtual Memory Management Unit (MMU), which was present in the previous STMicroelectronics processor family. The main features of the *core a* and *core b* are shown in Table 1. The *Eigher* processor belongs to the *A-line* devices.

	Eiger (core b)	Eiger (core a)	Bernina (core a)	Chorus (core a)
#pipeline	Dual issues	Dual issues	Dual issues	Single issues
#stage of pipeline	5	5	5	4
ISA	VLE, Book-E	VLE, Book-E, LSP	VLE, Book-E, LSP	VLE, Book-E
#GPR	32 registers (32 bit each)	32 registers (32 bit each)	32 registers (32 bit each)	32 registers (32 bit each)
#SPR	109 registers (32 bit each)	108 registers (32 bit each)	111 registers (32 bit each)	99 registers (32 bit each)
Data cache	4KB	-	8KB	-
Instruction cache	8KB	8KB	16KB	-
Debug unit	Nexus3	Nexus3	Nexus3	Nexus3
BTB	8 entries	8 entries	8 entries	4 entries
FPU	EFPU2	EFPU2	EFPU2	EFPU2
Performance monitor	71 events	69 events	71 events	57 events
MMU	-	-	-	-
MPU	24 entries configurable	24 entries configurable	24 entries configurable	-

Table 1: The features of the SPC58 cores

2) Bernina

The *Bernina* processor (SPC58NNx) [56] is a multicore processor equipped with 6MB of flash memory, 512KB of shared RAM and 128KB of local RAM for each core. Two AMBA BUSs are present in the *Bernina* processors. The processor is equipped with three cores of the same type (called *core a*, *core b* and *core c*), and additional two cores used in lockstep configuration. The *core a* is based on a dual issues pipeline with five stages each. The VLE, Book-E and LSP assembly instructions are available in Bernina. In this core, there are 8KB of L1 data cache and 16KB of L1 instruction cache. The main features of the *Bernina's* core are shown in Table 1. The *Bernina* processor belongs to the *A-line* devices.

3) Chorus

The *Chorus* processor (SPC582Bx) [56] is a single-core (*core a*) processor equipped with 1MB of flash memory and 96KB of RAM. A single AMBA BUS is present in the processor. *Core a* is based on a single issue with four stages each. The VLE and Book-E assembly instructions are available in *core a*. In this core, there are not data cache and instruction cache. The main features of the *core a* are shown in Table 1. The *Chorus* processor belongs to the *C-line* devices.

C. Compare the core and the units of the SPC58 family

This subsection compares the main features of the cores used in the processors previously described. In Table 2 the features of different units of the core are compared. The processor's development roadmap analyzed in Figure 9 is considered. In general, the units inside of the *Eiger core b* are classified as *reduction* compared to the *Eiger core a*. A similar consideration is possible for the units of the *Chorus core a* respected to the *Eiger core b*. Instead, the units of the *Bernina core a* are classified as *increases* compared with the *Eiger core a*. From Table 2, it can be seen that the Branch Target Buffer (BTB) unit remains almost unchanged for all processors, except for the *Chorus* where the number of entries is halved. The BTB is based on the Branch Instruction Cache (BTIC). It consists of a small cache memory contain two information: the address of the conditional jump instructions and the statistical prediction of the branch executed. The entries are populated and updated in accord with the Least-Frequently Used (LFU) approach, where the less frequently used entry is overwritten. The BTB has one SPR used to config the unit, to enable and disable the BTB, or to invalidate all entries. The BTIC implementation is architecturally transparent, so it does not have to be saved during the context switch.

In all the cores there is only one divider unit shared between the two pipelines. In the cores that manage the LSP instructions, the divider operates at 64 bits, in the other cores it operates at 32 bits. Table 2 shows the number of clock cycles needed to perform a single division. The number of clock cycles is variable, it depends on the implementation of the divisor unit and depends on the number of low logical bits present in the operands of the division. In the proposed architecture, the whole processor stalled while the division is performed. As the division unit, the multiplier unit is also shared between the two pipelines. However, each multiplication instruction is always executed in two clock cycles. In the cores that manage the LSP instructions, the multiplier is considerably more complex and it is able to perform a considerable number of different types of multiplication instructions. Moreover, in the cores with the LSP ISA instructions are management, the multiplier operates at 64 bits; while in the other cores it operates at 32 bits. It should be noted that two distinct multipliers are present internally in the ISA LSP cores. The first multiplier is used to managed the VLE multiplications instructions that perform 32 bits multiplication, while the second multiplier is used for LSP instructions. In the LSP instruction multiplier is also present a hardware

accumulator used by some particular instructions. A decoder unit is integrated into the multiplier, it is able to decode the LSP multiplication instruction. The register file consists of 32 generic registers (GPR) equal for all processors of the Power-PC architecture, while the number of special registers (SPR) depends on the number of peripherals and their functionalities. All the registers are physically implemented in 32 bits, but in the cores that handle the LSP instructions, there is the possibility of concatenating two registers to obtain a 64-bit logic register. The register file has three read ports and two write ports in the cores that implementing the VLE and Book-E instructions, while the ports are four in reading and four in writing in the cores that implementing the LSP instructions. In accord with the EABI standard [33], the R1 register is used as Stack Point. The R2 and R13 registers are used to access the Small Data Area (SDA) memory regions. The SDA regions are used managing the global variables and the global constants. The registers from R3 to R10 are volatile and they are used for passing the parameters to the functions written in C language. Moreover, the register R3 is used for the return value of the C functions. The volatile registers do not need to be saved in the stack frame during the context switch or when a function is called. Instead, the registers from R14 to R31 are non-volatile and must be saved in the stack frame.

The shifter unit is able to performed shift and rotate instructions to right or to left. It can operate on 32 bits register in cores that handle the VLE instructions, or 64 bits register in cores that handle the LSP instructions. Each pipeline is equipped with its own shifter unit. Similarly to the shifter units, the adder units are also duplicated. There is the adder unit for each pipeline. The adder performs the operations in one single clock cycle. Whit the VLE instructions the adder units performing operation at 32 bits. In the cores that manage the LSP instructions, the adder unit is adapted to work with data of 64 bits.

			BTB	DIVIDER	MULTIPLIER	REGISTER FILE	SHIFTER	ADDER
Eiger (*core a*)	Features		1) 8 entries 2) BTAC 3) LFU	1) 3-24 clock cycle for a division 2) 64 bits integer divider	1) Multiplier with 64 bits output 2) Multiplier with 32 bits input 3) Accumulator 4) Own decoder unit 5) Two internal multipliers: 16 bit and 32 bits	1) 32 GPR 2) 76 SPR 3) Concatenation of two registers for management the LSP instructions	1) 64 bits rotate and shift left/right operation	1) 64 bits and 32 bits adder
	#Instruction	LSP	-	1	457	0	16	24
		VLE	-	22	4	8	8	8
		Book-E	-	8	8	20	4	20
	#SPR used		1	-	-	-	-	-
Eiger (*core b*)	Features		1) 8 entries 2) BTAC 3) LFU	1) 6-16 clock cycle for a division 2) 32 bits integer divider	1) Multiplier with 32 bits output 2) Multiplier with 16 bits input	1) 32 GPR 2) 77 SPR	1) 32 bits rotate and shift left/right operation	1) 32 bits adder
	#Instruction	LSP	-	-	-	-	-	-
		VLE	-	22	4	8	8	8
		Book-E	-	8	8	20	4	20
	#SPR used		1	-	-	-	-	-
Bernina (*core a*)	Features		1) 8 entries 2) BTAC 3) LFU	1) 6-24 clock cycle for a division 2) 64 bits integer divider	1) Multiplier with 64 bits output 2) Multiplier with 32 bits input 3) Accumulator 4) Own decoder unit 5) Two internal multipliers: 16 bits and 32 bits	1) 32 GPR 2) 79 SPR 3) Concatenation of two registers for management the LSP instructions	1) 64 bits rotate and shift left/right operation	1) 64 bits and 32 bits adder
	#Instruction	LSP	-	1	457	0	16	24
		VLE	-	22	4	8	8	8
		Book-E	-	8	8	20	4	20
	#SPR used		1	-	-	-	-	-
Chorus (*core a*)	Features		1) 4 entries 2) BTAC 3) LFU	1) 7-35 clock cycle for a division 2) 32 bits integer divider	1) Multiplier with 32 bits 2) Multiplier with 16 bits	1) 32 GPR 2) 67 SPR	1) 32 bits rotate and shift left/right operation	1) 32 bits adder
	#Instruction	LSP	-	-	-	-	-	-
		VLE	-	22	4	8	8	8
		Book-E	-	8	8	20	4	20
	#SPR used		1	-	-	-	-	-

Table 2: The features of the different units inside of the cores

V. Experimental results

This section reports the experimental results for the SPC58 processor family produced by STMicroelectronics. The features of the family and the features of its processors have been described in Section IV. In particular, the first subsection reports the Fault Coverage figures for the different units of the cores of the different processors, in accord with the proposed porting methodology described in Section III. Furthermore, the development times of the STLs and the fault-simulation times are also reported. Later, some interesting examples of the reduction units are discussed and analyzed. Afterwards, the efficiency of the ATPG-base and Evolutionary-base approaches are discussed. Finally, the ISA contribution is analyzed.

A. Portable STL results

In order to demonstrate the effectiveness of the proposed methodology discussed in Section III, the FC values obtained using the same portable test programs on the different processors are reported. In particular, the results for BTB, Divider, Multiplier, Register File, Shifter, and Adder are analyzed.

1) BTB

A possible BTB test methodology is proposed in [28]. In [28] the test is performed by performing a sufficient number of conditional branches used to load the BTB entries. After that the BTB has been initialized, a new sequence of branches is performed to verify the BTB prediction. The test methodology proposed in [28] is portable because it is developed with the *deterministic* approach. It considers the BTB to a functional point of view and it is independent of its synthesis. The structure of the test programs is shown in Figure 10. The test program is implemented considering the sizes of the different RAM memories present in the different processors, the space available for each test program is shown in Figure 10. The algorithm proposed in [28] is adapted to the memory size of each processor. The numbers of possible permanent stuck-at faults for each BTB unit of the different core are shown in Table 3, while the Fault Coverage figure for each processor is shown in Table 4. The first column, called *FC*, reports the Fault Coverage obtained considering all possible BTB faults. The second column (*FC increm.)* considers the incremental fault-simulation approach. In the incremental approach, the faults previous detected by other test programs are not again considerate during a new fault-simulation campaign. The last column reports the time necessary to perform the fault-simulation of the single test program considering all faults present in the UUT. The *FC total* value report the fault coverage obtained considering the union of all test programs. While the *FC total and cascade* value report the final FC. *FC total and cascade* value also includes the contribution of non-portable test programs eventually implemented. It is possible to note that no other test program introduces a cascade phenomenon on the FC of the BTB, and no other non-portable test programs are implemented for the BTB unit. Analyzing the FC values obtained with the same test program on all processors, it is possible to note a constant FC value around 70% for the *Eiger core a*, the *Eiger core b* and the *Bernina core a*, as shown in Table 4. It is possible to see from Table 2 that the BTBs unit of these three cores have the same features. It should be noted that at the GATE level these three BTBs are different because the number of faults is significantly different as shown in Table 3.

	Eiger (core a)	**Eiger (core b)**	**Bernina (core a)**	**Chorus (core a)**
BTB	21,434	16,506	19,892	11,037
Divider	34,013	18,928	35,018	20,337
Multiplied	73,638	29,717	63,962	32,868
Register file	146,217	82,518	140,654	68,217
Shifter	12,422	3,132	17,115	3,136
Adder	14,762	6,446	19,758	3,972

Table 3: Number of faults for each unit of each core

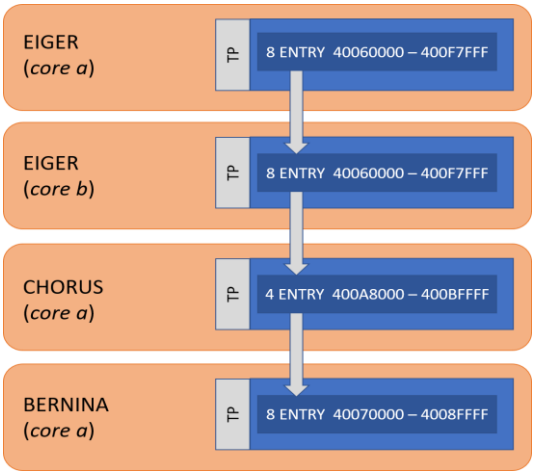

Figure 10: BTB test program structure

The BTB unit of the *Chorus core a* is classified as REDUCED, compared to the BTB of the other cores considered in this family. In the *Chorus core a,* the number of BTB entries is halved. Only for this core, it is necessary to modify the test program. It is necessary to reduce the number of branches performed to initialize the BTB, and the number of branches performed to verify the predictions of the BTB. The test method proposed in [28] is easily scalable with respect to the number of

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
BTB1	63.38	+63.38	168	57.76	+57.76	195	63.40	+63.40	227	53.43	+53.43	98
BTB2	50.71	+6.24	161	45.65	+9.82	174	54.98	+6.20	206	42.13	+7.06	88
BTB3	27.41	+2.28	85	14.81	+1.92	92	19.54	+1.34	161	18.54	+3.83	41
FC total		71.90			69.50			71.16			64.32	
FC total and cascade		71.90			69.51			71.16			64.33	

Table 4: The BTB case

BTB entries. The FC reduction is due to the decrease of the memory available in the *Chorus* processor, as discusses in sub-section B.3 of the Case Study. The variety of the memory addresses that can be loaded in the BTB entries is considerably lower in the *Chorus* processor.

2) DIVIDER

The test programs for the division unit consider divisions executed between the checkerboard patterns (0x0000, 0xFFFF, 0xAAAA, 0x5555, 0xCCCC, 0x3333) and numbers of power of two (2, 4, 8, 16, ...). The use of these generic patterns does not require the calculation of ATPG patterns valid only for a specific implementation of the divisor, as discussed in Section III. The structure of the test programs is shown in Figure 11. Two sub-tests have been implemented, one considering the VLE instructions on 32-bit operands and one considering those with 64-bit operands. Furthermore, a non-portable test program was implemented using the ATPG approach.

The Fault Coverage value obtained on the different processors of the family under examination is about 78% (77% *Eiger core a*, 83% *Eiger core b*, 76% *Bernina core a*, 77% *Chorus core a*), as shown in Table 5. In the *Eiger core a* and *Bernina core a*, the test methodology is extended to 64-bit patterns using LSP instructions. On the other cores, the VLE instructions with 32-bit patterns are used. It is interesting to note that the DIV3 test

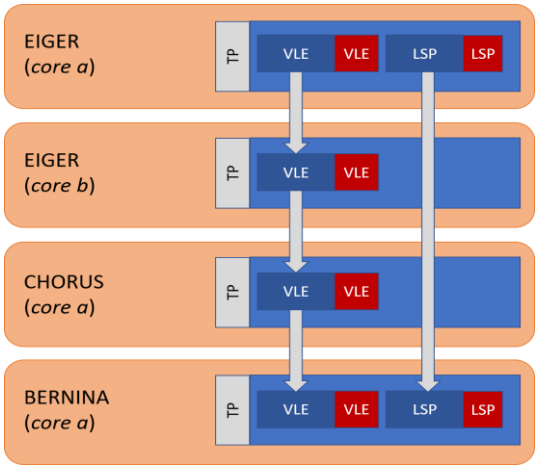

Figure 11: Divider, multiplier, shifter, and adder test programs structure

program on the *Chorus* processor does not give a useful contribution to the final FC, this test program was discarded in the final STL. In order to increase the FC of the dividers, some non-portable test programs are implemented with the ATPG approach. The contribution of these test programs is indicated by comparing the FC total value with the FC total and cascade value of Table 5.

3) MULTIPLIER

As introduced in section IV.C, the multiplier unit for the *Eiger core a* and *Bernina core a* acquires many features compared to the *Eiger core b.* Moreover, the multiplier of the *Eiger core a* has similar features to the multiplier of the *Chorus core a*. The structure of the test programs is shown in Figure 11. Two sub-tests have been implemented, one considering the VLE instructions on 32-bit operands and one considering those with 64-bit operands. Furthermore, a non-portable test program was implemented using the ATPG approach. The first set of test programs developed to test the *Eiger core a* multiplier is developed. Afterwards, a second set of test programs able to detect the faults present in the new version of the multiplier is implemented. The features of the two versions of the multiplier are shown in Table 2, while the number of faults is shown in Table 3. It is possible to note that the number of the possible faults present in the multipliers of the *Eiger core a* and *Bernina core a* is doubled compared to the number of the possible faults present in the multipliers of the *Bernina core b* and *Chorus core a*. This remarkable difference in the number of faults is due to the hardware added in the LSP version of the multiplier. In analogy to the divider unit, the multiplier unit is also tested with a checkerboard pattern and numbers of power of two. Different VLE and LSP instructions are used to apply the test patterns. From Table 9, it is possible to notice an FC of about 90% on the multiplier (87% *Eiger core a*, 91% *Eiger core b*, 92% *Bernina core a*, 91% *Chorus core a*). Furthermore, it can be seen a considerable influence of the LSP test program on the FC total value. The impact of the LSP test program is around 28% on FC total value, the LSP test program is present only in the LPS version of the multiplier for the *Eiger core a* and *Bernina core a*. Some test programs with the ATPG approach are developed to increase the FC of multipliers on different processors. The influence of these additional non-portable test programs is about 5%.

4) REGISTER FILE

The Register File of these processors is internally divided into numerous subunits. The C*ontrol Register* sub-unit contains the status registers and control registers of the core, the *Decode* sub-unit contains the logic able to address each register, the *Exception Register* sub-unit contains the status register and

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
DIV1	41.96	+41.96	111	60.10	+60.10	49	39.09	+39.09	65	55.88	+55.88	58
DIV2	57.96	+19.16	114	42.24	+3.02	48	53.33	+19.86	66	64.25	+14.45	62
DIV3	64.08	+10.69	109	45.37	+2.08	47	57.91	+9.81	62	50.24	+0.14	61
DIV4	59.29	+2.00	104	76.34	+2.27	48	51.88	+6.41	63	56.27	+4.85	62
MUL_DIV	37.00	+3.84	123	66.80	+16.04	63	28.72	+1.66	91	46.93	+2.13	75
FC total		77.65			83.51			76.83			77.31	
FC total and cascade		79.23			86.78			78.01			82.20	

Table 5: The DIVIDER case

interrupt management registers used during an interrupt request. The G*eneral Register* sub-unit contains the 32 GPR registers, while the *Mem Mux* sub-unit contains the Register File logic interface. In [27] a possible test methodology for the Register File is discussed. The methodology proposed in [27] is able to detect the faults present in some of the sub-units of the Register File, in particular in the *General Register.* The *Control Registers* sub-unit, the *Decode* sub-unit, and the *Mem Mux* sub-unit are partially tested in [27]. The test methodology proposed in [27] for the Register File is based to write and to read the checkerboard pattern in all registers. The registers are subdivided into two groups. In each group, the encoding of all registers has a hamming distance higher than one bit with respect to each other. The structure of the test program for the register file is shown in Figure 12. The results of Table 10 show the overall FC of the Register File, while the results of Table 6 and Table 7 shows the FC for its sub-units. Table 6 and Table 7 considering two different test programs set, Table 6 considering only the RF_MEM test program, while Table 7 considering the whole STL. The methodology proposed in [27] is implemented in the RF_MEM test program.

Other test programs have a great influence on the final FC of the Register File, as can be seen by comparing Table 6 with Table 7. Table 8 shows the number of faults for each sub-unit of the Register File. Only the *General Register* sub-unit is considered for the porting approach. The methodology proposed in [27] is independent by the implementation of the Register File, it is portable from one processor to another processor of the same processor family; this methodology can be used, in another non-portable test program, also for testing the SPR registers classified as *exclusive.* It is possible to note in Table 7 that the FC figure of the General Register sub-unit of the Register File maintains a good FC value on all cores, the test program used in all cores is always the same.

Register File sub-unit	**Eiger (core a)**	**Eiger (core b)**	**Bernina (core a)**	**Chorus (core a)**
Control Register	68.98%	65.27%	67.17%	66.80%
Decoder	73.59%	69.23%	74.06%	74.84%
Exception Register	3.25%	2.89%	3.14%	2.97%
General Register	99.98%	91.48%	94.26%	98.50%
Mem Mux	79.45%	82.68%	77.46%	81.28%
Glue logic	97.16%	98.58%	98.12%	98.61%
FC total	59.95%	62.54%	60.07%	70.21%

Table 6: The FC obtained by the RF_MEM test program on the Register File sub-units

Register File sub-unit	**Eiger (core a)**	**Eiger (core b)**	**Bernina (core a)**	**Chorus (core a)**
Control Register	97.43%	90.27%	92.90%	91.15%
Decoder	100.00%	94.04%	99.13%	92.80%
Exception Register	89.14%	80.08%	85.55%	100.00%
General Register	99.98%	91.48%	94.26%	98.50%
Mem Mux	97.13%	97.47%	98.79%	95.99%
Glue logic	99.89%	100.00%	98.97%	99.53%
FC total	97.43%	90.27%	97.15%	92.90%

Table 7: The FC obtained by the whole STL on the Register File sub-units

Register File sub-unit	**Eiger (core a)**	**Eiger (core b)**	**Bernina (core a)**	**Chorus (core a)**
Control Register	3,129	2,253	2,708	2,065
Decoder	2,546	1,308	3,042	924
Exception Register	15,914	10,337	15,181	9,410
General Register	50,864	20,762	52,579	23,176
Mem Mux	69,414	45,508	63,826	31,218
Glue logic	4,350	2,350	3,318	1,424
FC total	146,217	82,518	140,654	68,217

Table 8: The number of faults for each sub-unit of the Register File

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
MUL	55.21	+55.21	144	90.28	+90.28	84	53.04	+53.04	255	89.62	+89.62	89
MUL_DIV	40.02	+2.79	237	62.77	+1.49	119	37.32	+10.76	341	59.74	+1.50	132
MUL_LSP	48.16	+29.21	156	-	-	-	49.53	+28.49	278	-	-	-
FC total		87.21			91.77			92.29			91.12	
FC total and cascade		91.76			98.23			92.30			97.88	

Table 9: The MULTIPLIED case

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
RF_MEM	59.98	+59.98	659	62.54	+62.54	541	60.07	+60.07	504	70.21	+70.21	498
FC total		59.98			62.54			60.07			70.21	
FC total and cascade		97.43			90.27			97.15			92.90	

Table 10: The REGISTER FILE case

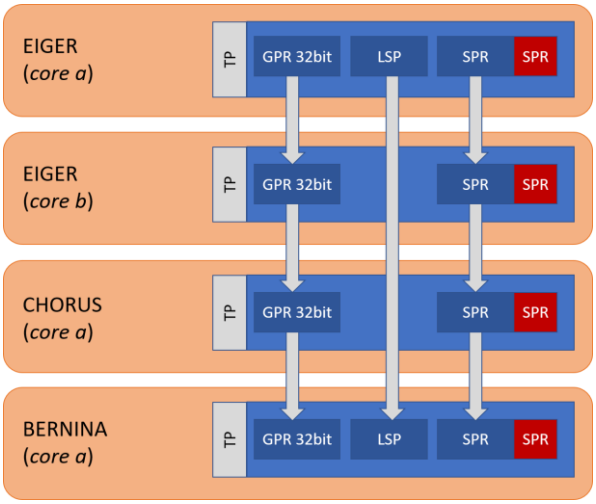

Figure 12: Register file test program structure

5) SHIFTER

The same considerations of the divider unit and the multiplier unit can be replicated for the shifter unit. Some shift operations with checkerboard patterns are performed on 32-bit and 64-bit registers. The VLE instructions are used on the 32-bit registers in the *Eiger core b* and *Chorus core a*, while the LSP instructions are used on the 64-bit registers in the *Bernina core a* and *Chorus core a*.

The structure of the test program for the shifter is shown in Figure 11. The FC obtained from the same test program on all the cores being examined is around 84%, as it shows in Table 11 (80% *Eiger core a*, 85% *Eiger core b*, 82% *Bernina core a*, 89% *Chorus core a*). Also for the shifter unit, other non-portable test programs are implemented with the ATPG approach. Considering also the non-portable test programs' contribution, a high FC on the shifters is reached.

6) ADDER

The test methodology proposed for adder units is based on the sum of checkerboard test patterns. Furthermore, some sum operations between the very large numbers and the small numbers are implemented to detect the faults present in the adder's carry paths. The structure of the adder test program is shown in Figure 11. The FC obtained on the adder units with the same test program is about 85%, as it shows in Table 12 (87% *Eiger core a*, 90% *Eiger core b*, 88% *Bernina core a*, 84% *Chorus core a*). The FC of the adder units is influenced by a high cascade effect given by the other test programs. This benefic effect on the FC is due to how the signature is constructed in the test programs. As discussed in Section B.2 of the Background, the signature is obtained by accumulation, i.e. sum operation, of the partial results generated during the test program. These sum operations contributed to the detection of the faults present in the adder, the effect of the cascade on the FC is shown in Table 12.

B. Other Test programs for SHARED units

With reference to the portable classification of the units discussed in Section III.B, it is possible to classify the Logic Instruction unit and the Exception Control unit as *SHARED*. The functionalities of the Logic Instruction unit and the functionalities of the Exception Control unit remain equal for all the processors of the SPC58 family. Two portable test programs can be developed for the Logical unit and for the Exception Control unit. The test program able to detect the faults present in the Logic Instruction unit uses AND, OR, NOT and XOR logic operations between checkerboard patterns.

The Exception Control unit is tested as described in [5]. The methodology proposed in [5] consists of voluntarily triggering interruptions and verifying that the correct Interrupt Service Routine (ISR) is performed for each interruption source.

The test methods proposed for the Logic Instructions unit and for the Exception Control unit are developed with the *deterministic* approach, these methodologies are independent of the synthesis at the gate level of the units. As it is possible to see from Table 13 and Table 14, the same test programs developed for a single processor of the family have been used on all processors of the family. The FC values of the Logic Instruction unit and of the Exception Control unit remains almost unchanged on all different cores.

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
SHIFTER	80.90	+80.90	83	85.26	+85.26	32	82.01	+82.01	47	89.48	+89.48	21
FC total		80.90			85.26			82.01			89.48	
FC total and cascade		83.04			99.69			89.03			99.59	

Table 11: The SHIFTER case

	Eiger (core a)			**Eiger (core b)**			**Bernina (core a)**			**Chorus (core a)**		
TEST PROGRAMS	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]	FC [%]	FC increm. [%]	Time [Hour]
ADDER1	87.53	+87.53	38	90.63	+90.63	14	88.69	+88.69	49	84.89	+84.89	9
FC total		87.53			90.63			88.69			84.89	
FC total and cascade		93.46			94.41			93.56			93.25	

Table 12: The ADDER case

	Eiger (core a)		Eiger (core b)		Bernina (core a)		Chorus (core a)	
	#Faults	FC[%]	#Faults	FC[%]	#Faults	FC[%]	#Faults	FC[%]
Program Counter	37,793	70.85	20,411	66.25	32,113	66.49	20,203	65.95
Divider	34,013	79.23	18,928	86.78	35,018	78.01	20,337	82.20
Logic Instruction	4,172	89.94	3,232	95.08	3,360	93.74	2,288	92.48
Multiplier	73,638	91.76	29,717	98.23	63,962	92.30	32,868	97.88
Shifter	12,422	83.04	3,132	99.69	17,115	89.03	3,136	99.59
Exception Control	15,297	50.63	10,363	47.54	13,695	55.28	9,834	55.37
BTB	21,434	71.90	16,506	69.51	19,892	71.16	11,037	64.33
Register File	146,217	97.43	82,518	90.27	140,654	97.15	68,217	92.90
Fetch Unit	47,460	85.76	25,622	880.52	39,373	71.63	15,782	63.53
Forward	134,091	75.85	44,679	78.67	115,287	77.65	29,632	87.39
Decode Unit	66,566	53.28	30,141	69.61	101,900	67.29	19,005	63.65
Load/Store Unit	17,879	72.51	10,631	71.89	22,712	82.83	15,669	73.05
Brinc Unit	1,296	95.91	-	-	1,338	92.37	-	-
Merge Unit	4,044	90.48	-	-	4,136	91.32	-	-
Saturate Unit	-	-	-	-	16,404	73.00	-	-
Adder Unit	14,762	93.46	6,446	94.41	19,758	93.56	3,972	93.25
Control Logic	50,309	74.82	28,591	78.71	58,202	73.01	19,463	70.64
Performance Monitor	2,620	87.90	880	100	3,096	86.82	564	90.25
Glue Logic	31,216	72.90	13,204	77.45	21,507	68.45	10,363	79.07
TOTAL	715,229	80.07	345,001	80.98	729,522	80.26	282,370	80.40

Table 13: The final FC of each processor

	Eiger (core a)			Eiger (core b)			Bernina (core a)			Chorus (core a)		
	#TP	Memory occupation [Bytes]	Duration [C.C.]	#TP	Memory occupation [Bytes]	Duration [C.C.]	#TP	Memory occupation [Bytes]	Duration [C.C.]	#TP	Memory occupation [Bytes]	Duration [C.C.]
Program Counter	0	-	-	0	-	-	0	-	-	0	-	-
Divider	8	6,280	11,600	8	4,668	10,479	8	6,280	11,600	8	6,280	11,600
Logic Instruction	1	690	350	1	690	350	1	690	350	1	690	350
Multiplier	5	3,800	2,700	5	2,900	2,100	5	3,800	2,700	5	2,900	2,100
Shifter	2	2,200	2,900	2	720	800	2	2,200	2,900	2	720	800
Exception Control	5	8,100	28,000	5	8,100	28,000	5	8,100	28,000	5	8,100	28,000
BTB	3	3,000	32,700	3	3,000	32,700	3	3,000	32,700	3	2,600	7,438
Register File	2	4,100	2,650	2	4,100	2,650	2	4,100	2,650	2	4,100	2,400
Fetch Unit	1	2,700	2,315	1	2,700	2,315	1	2,700	2,315	1	1,800	1,980
Forward	1	1,600	1,540	1	1,600	1,540	1	1,600	1,540	1	978	1,100
Decode Unit	3	7,000	33,550	3	7,000	33,550	3	7,000	33,550	3	2,800	21,200
Load/Store Unit	1	920	1,200	1	920	1,200	1	920	1,200	1	920	1,200
Brinc Unit	1	1,300	820	0	-	-	1	1,300	820	0	-	-
Merge Unit	2	4,400	3,350	0	-	-	2	4,400	3,350	0	-	-
Saturate Unit	2	-	-	0	-	-	2	2,190	5,180	0	-	-
Adder Unit	2	1,600	1,300	2	1,450	1,200	2	1,600	1,300	2	1,150	860
Control Logic	0	-	-	0	-	-	0	-	-	0	-	-
Performance Monitor	2	3,000	2,700	2	3,000	2,700	2	3,000	2,700	2	2,600	2,500
TOTAL	41	49,690	127,675	36	40,848	119,584	41	49,880	132,855	36	35,638	92,328

Table 14: The STL features of each core

	Eiger (core a)		Eiger (core b)		Bernina (core a)		Chorus (core a)	
	#IST-TP	IST-C [%]	#IST-TP	IST-C [%]	#IST-TP	IST-C [%]	#IST-TP	IST-C [%]
Book-E	7	94.88	7	94.88	7	94.88	7	94.88
VLE	8	94.81	8	94.81	8	94.81	8	94.81
LSP	9	94.55	-	-	9	94.55	-	-
TOTAL	24	94.56	15	94.85	24	94.56	15	94.85

Table 15: The ISA test programs

C. Chorus core a single issue case

This paragraph is dedicated to the *Chorus* processor. This processor is obtained from the *Eiger* processor, as shown in Figure 9. In particular, the *Chorus core a* is obtained removing some features by the *Eiger core b*, as discussed in Section IV. This processor, belonging to the low power line of the SPC58 family, consists of a single pipeline and a limited number of peripherals. The porting phase is considerably simplified for the *Chorus* processor due to the presence of only one pipeline in the *Chorus core a*. All the test programs in the dual issue processors are implemented to apply the same test patterns to both pipelines. For example, the sum operations that performed the test to the adder unit are performed twice; the first time to test the adder unit of the first pipeline and a second time to test the adder unit of the second pipeline [27][57]. Therefore, in the test programs, it is possible to disable the replicas of the sub-tests associated with the test of the unit of the second pipeline. As the reader can see from the final FC results reported in Table 13, the removal of the sub-test replicas does not affect the final FC of the internal units of the *Chorus* processor. The FC values of the units of the *Chorus core a* remain in line with the FC values of the other processors. However, it has a significant decrease in the number of test programs and their duration, as shown in Table 14.

D. Fault Coverage results discussion

Overall, the Tables 4-5-9-10-11-12 show how the proposed porting methodology is effective for processors belonging to the same family. Furthermore, it suggests that the development of the test programs in a portable way is more efficient. In other words, the test programs are developed respect the STL Development Plan (SDP) and the test program structure identified with the proposed approach.

All the experiments were performed on a server equipped with two XEON ES-2620V3 processors operating at 2.4GHz with 64GB of RAM available. All the fault-simulations are performed with 15 parallel processes. The time required to execute each fault-simulation has been reported in hours in each table. Table 13 shows the detail about the FC and the number of faults present in each unit and for each core. The last row reports the total number of faults present in the core and the final FC of the whole core. Instead, Table 14 shows for each unit of each core the number of test programs developed (#TP), the memory occupation and the duration in clock cycle.

E. Developed time

Figure 13 shows the development times of the STLs for the considered cores. As discussed in Section II.D and in Section III.C, it is possible to develop STL only once the gate-level synthesis is complete. The STL development starting when the gate level synthesis is completed and the verification phase of the physical device is passed. While the development of the STL ends when the FC of the core is at least 80%. This value is calculated by the microcontroller manufacturer in accordance with the ASIL D ISO26262 standard [58].

Interestingly, about 7 months are needed to develop the STL for the first core. Subsequent processors require less development time because they benefit from portable tests developed for the previous processors. In general, for each new processor remains necessary to develop some test programs for the units classified as *exclusive*. Moreover, the non-portable test programs required to be developed for increasing the FC; typically, new ATPG test patterns must be generated for the non-portable test programs. Non-portable tests are used to fill the small FC drop if present. This is particularly evident in the case of the *Chorus* processor; the *Chorus* STL is obtained by disabling many of the STL sub-tests developed for the *Eiger core b*. The times reported in Figure 13 consider the fault-simulation times, and the processor setup time, i.e. the time required to configure the development environment for the processor. Figure 14 also shows the period of introduction of the processor on the market. This period represents a deadline defined by company marketing. It can be seen that the development of the STLs falls within these deadlines.

Considering the ISO26262, the Single-Point Fault Mode (SPFM) is used for the single permanent stuck-at fault. The 80% threshold was calculated by the manufacturer considering the silicon area surface of the core respect to the total surface of the *device*. A 99% FC on the overall microcontroller is finally obtained with the combination of STL with different approaches (like ECC, lockstep or other Hardware- and Software-based safety mechanisms) applied to all the components included in the microcontroller. In this paper, only the development of STLs for the processor has been considered. Given that to reach ASIL D levels, the processor cores have to run in Lockstep configuration, the STL approach is indispensable to intercept misbehaviors at early time, before their effects are leading to a processor failure noticed by the lockstep. At microcontroller level, it is also correct to consider that the processor is just a minor part of the silicon area, and final ASIL numbers are also related to the other modules, such as the embedded memories and peripherals. Typically, the memories are tested resorting an ECC, while the peripherals can be tested with dedicated Hardware-based or Software-based approaches, as discussed in [59], [60], [61].

Figure 14 shows a projection of STL development using the strategy proposed in [5]. In Figure 14, it is possible to see that the *core b* of *Eiger* and *Bernina* do not respect the deadlines.

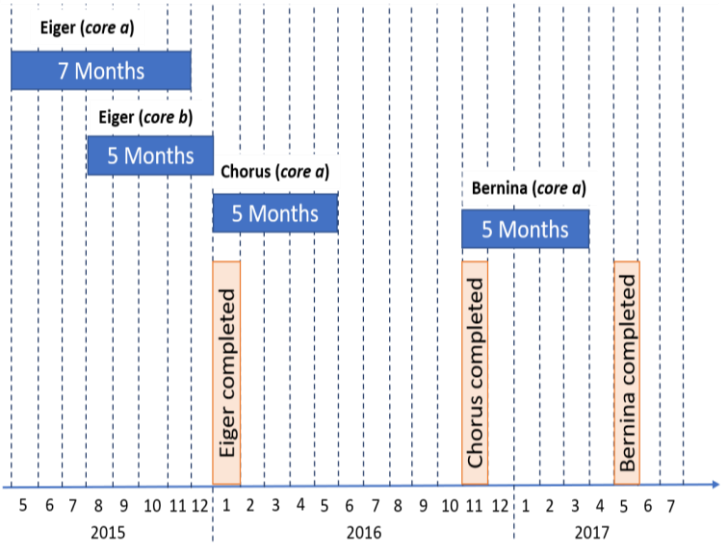

Figure 13: Portable STL developed time

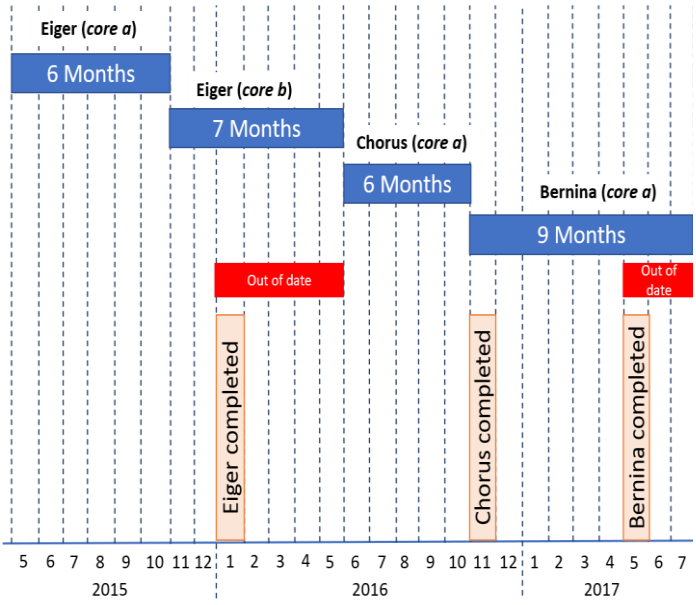

Figure 14: STL developed time with the approach proposed in [5]

Comparison between ATPG-base and Evolutionary-base approaches.

In order to demonstrate the inefficiency of the ATPG approach in terms of test program portability, some non-portable test programs developed for the *Eiger core a* was fault-simulated on the other cores of other processors. For the same purpose, also some test programs developed with the evolutionary approach are assessed on other processors. The test programs developed for the divisor and multiplier units are considered. Table 16 shows the FC results obtained for the divider unit and for the multiplier unit. The test programs have been developed for the *Eiger core a*; afterwards, the same test programs are evaluated on the other cores

It is possible to see from Table 16 that the test programs developed with the two non-portable approaches introduce a good fault coverage value only on the core for which they were developed; there is a significant loss of FC on the other cores.

	Eiger (core a)	**Eiger (core b)**	**Bernina (core a)**	**Chorus (core a)**
Divisor unit				
ATPG	65.42%	23.58%	12.59	26.02%
Evolutionary	68.50%	15.89%	9.57	21.64%
Multiplier unit				
ATPG	70.87%	28.15%	22.06%	27.59%
Evolutionary	75.57%	19.57%	19.89%	18.61%

Table 16: ATPG-based approach and the evolution-based approach

F. ISA coverage

As discussed in Subsection B.2 of the background, the Instruction Self-Test is an alternative metric used for the functional testing of the processors. This metric is based on the execution of all ISA instructions at least once. The IST test programs are easily portable because they are developed in a functional way. Table 17 shows the number of IST Programs (#IST-TP) developed for each processor and the related IST Coverage (IST-C). The purpose of this section is to show the contribution of the IST programs on the processor's FC. Two experiments are performed for this analysis; the first experiment performs a comparison between the IST Program and the stuck-at Test Program concerning the Logical Instructions unit. The second experiment compares the FC obtained on the Decoder unit considering all the IST test programs and all test programs developed to detect the stuck-at faults. The experiments were performed on the *Bernina core a*.

1) Logic Instruction unit experiment

In the first experiment, the IST program on logic instructions is fault-simulated on the whole core. Followed, the test program developed to detect the stuck-at faults on the Logic Instructions unit is fault-simulated on the whole core. The two fault-simulation results are compared, and the intersections between the different groups of faults are reported in Figure 15.

The number of possible permanent stuck-at faults of *Bernina core a* is 729.522 (as indicated in Table 15), the developed STL provides a FC of 80.26% (equal to 585.514 faults detected). Figure 15 shows also the number of faults detected by the *Logic instruction unit test program* and by the *IST logic instruction test program*. It is possible to see that all the faults detected by the *IST logic instruction program* are also detected by the STL.

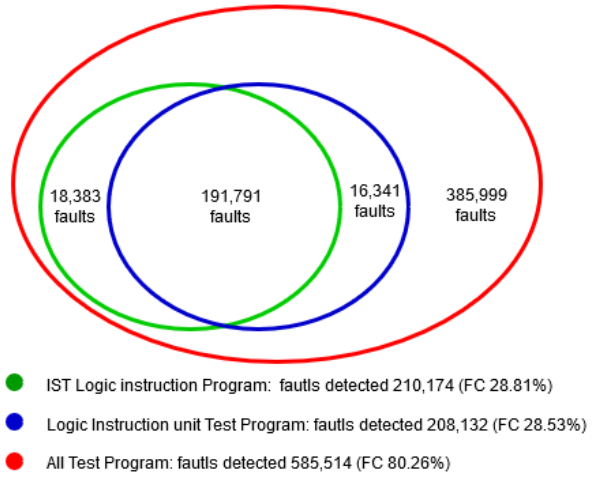

Figure 15: ISA Logic Instruction unit results

Figure 15 provides also a comparison between the *Logic instruction unit test program* and the *IST logic instruction program*. The great majority of the faults are detected from both programs. However, there are 18,383 faults detected from the IST program. In any case, the STL detected also the faults detected from the IST test program. This first experiment demonstrates the ineffectiveness of IST programs to detect permanent stuck-at faults.

These results were performed using the Fault List Analyzer Tool (FLAT) [36] which allows of comparing the results of different fault simulations.

2) Decoder unit experiment

In the second experiment, two fault-simulations are performed on the Decoder unit with two different sets of programs. The first set includes all IST programs; while the second set includes only the test programs able of detecting the stuck-at faults in the Decoder unit. For the *Bernina core a*, three test programs have been implemented for the Decoder unit, as indicated in Table

16. Among the many possible approaches to test the Decoder unit, the test programs developed are implemented with the approach proposed in [29]. In [29] the legal and the illegal instructions are considered to detect the decoder's faults.

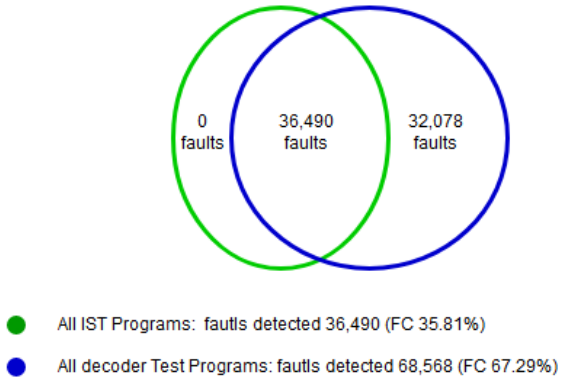

Figure 16: ISA Decoder unit results

From Figure 16, it is possible to notice that all the faults detected by the IST programs are already detected by the three specific test programs for the Decoder unit of *the Bernina core a*. Therefore, the IST programs do not introduce any effect on the FC of the Decoder unit. Furthermore, this experiment shows that the IST metric is not able to detect all the possible stuck-at faults present in the Decoder units.
The two experiments performed to demonstrate the ineffectiveness of the IST approach. The IST test programs do not introduce a real contribution to the final FC of the STL, these programs can be disabled.

VI. Issues about the industrial cases

This section reports some problems encountered during the development of the STL and some problems encountered during the integration of the STL with the software environment of the customers. Furthermore, some precautions and checks actuated during the development of the test programs are explained in this section.

A. Memory RAM used

As discussed in Section B.2 of the Background, the test programs can be classified as *intrusive* and *non-intrusive* [32]. With reference to RAM memory, *non-intrusive* test programs do not write or read from the RAM memory, while *intrusive* tests access to the RAM memory. The intrusive test programs are executed at boot-time, before the operating system are executed. Often in the automotive sector, the ECUs are never really switched off when the vehicle is switched off, but the ECUs are placed in a low power state. When the vehicle switched on, the ECUs return in execution state without a real restart. There is not the real boot phase of the OS. Therefore, the intrusive test programs are executed in the presence of data in RAM memory, these data must not be corrupted or altered. The RAM memory regions used by the test programs are known, the integrity of these regions is management by the customer's software environment during the *intrusive* test execution.

B. Multicore processor

Many processors of the SPC58 family operated in a multicore scenario, and some of them used different core types. This aspect introduced the problem to execute the same STL in parallel on different cores of the same type. Also, the problem of executing different STL libraries, for different core types, in the same processor is present. The different STLs must not be influenced by each other, in particular about the use of the RAM memory. A possible approach to parallelize the STLs avoiding the RAM memory conflicts is proposed in [62][63].

C. External debugger

The Nexus debug unit of the SPC58 family processors can work in two different operating modes; External Debug Mode (EDM) when the external debug is connected, usually via the JTAG port, or in Internal Debug Mode (IDM) when the external debug is not connected. The IDM configuration is used in many of the test programs because the IDM allows to test the interrupt management unit. The Performance Monitor, inside of the Nexus [52][56][64] , is used to increases the observability of the faults. When the external debug is connected, the test programs that use the Nexus in IDM must be disabled. An automatic control has been implemented to verify the presence of the external debugger and to disable the tests that used the Nexus. There is no loss of the ability of the STL to detect the faults by disabling these tests since the external debug is connected by the human operator in the workshop for the vehicle maintenance checks. On the field, the external debug is not connected and all the tests are normally performed.

D. STL optimization

At the end of the development of the STL, it is possible to perform an optimization of the tests. The STL can be optimized respect to different parameters; in particular, with respect to the FC, the STL time execution, and its occupation in the flash code memory. A special tool, called Fault List Analyzer Tool (FLAT) [36], has been used to analyze and optimize the STL. The FLAT considers different quality indexes able to evaluate the individual test programs and find an optimal solution, with respect to the parameters to be optimized. The FLAT use and its operations are illustrated in [36].

E. STL verification check and final test

During the development of the test programs, it is necessary to verify the EABI standard complied. Moreover, it is necessary to have relevance about the RAM memory locations used by the test program. The Monitor tool is implemented to verify the correct saving and recovery of the core registers. Moreover, the Monitor tool keeps track of the writing operations in the RAM memory executed by each test program. The Monitor tool is used during the development of the test program; it is activated during RTL logic simulations before invoking the test program and immediately after its return. When the Monitor tool is called, it stopping the logic simulation. The GPR registers and the whole RAM memory are dumped. The dump is executed before invoking the test program and when the test program return; these two dumps are compared. In the comparison phase, the presence of different values in the registers indicates an error in the EABI stack frame of the test program.

F. Test program signature building

The strategy used to build a test program signature has a significant impact on the performance and the effectiveness of the test program. In the literature, several strategies have been proposed. In this subsection, we briefly discuss the two most used. The strategy based on the Multiple-Input Shift Register (MISR) is computationally very expensive, if implemented via software, but it has a low aliasing. The aliasing is the situation in which a test program running on a defective unit produces a signature equal to the expected. This phenomenon occurs due to an escape of one or more faults, as discussed in [65], [66], [67]. In other words, some faults detected by the test patterns are masked during the construction of the signature. So due to aliasing, some potentially detectable faults are not detected by the test program. As discussed in [68], the MISR strategy is composed of a Flip-Flop chain alternating with numerous X-OR logic gates. This algorithm can be easily implemented in hardware, but requires many logical steps to be implemented in software. For this reason, it is typically used in hardware-based testing approaches.

The second possible strategy considered is based on the sum of the partial results obtained during the test. In [67], a possible hardware implementation is discussed; however, this algorithm can be easily implemented in software using subsequent sums operations. Compared to the MISR approach, the sum and accumulation approach is much less computationally expensive, but has higher aliasing, as discussed in [66]. However, for very short test programs that include few test patterns, the aliasing introduced with the accumulation strategy is negligible [66]. The portable test programs, described in this paper were implemented exploiting the accumulation strategy.

VII. Conclusion

The paper offers a wide background on the different in-field and on-line testing methodologies used in the modern industrial processors, with particular emphasis on the STL approach. Subsequently, the paper analyzes the problem of developing different STLs for different processors of the same family. An approach for developing portable test programs is proposed in this paper. The first aim of the proposed approach is to reduce the loss of Fault Coverage due to porting a test program from one processor to another of the same family. The second aim of the proposed approach is to reduce the development time of the test programs. To demonstrate the effectiveness of the proposed approach, it is applied to a real industrial case study. In particular, on the SPC58 family processors developed by STMicroelectronics for automotive safety-critical applications. This paper does not consider the development of STL for processors belonging to different processor families. In general, processors of different families do not have common features that allow an easy porting of test programs. Moreover, some considerations regarding the use of the Instruction Self-Test metric are reported. Finally, some practical considerations related to industrial development problems have been reported and analyzed. Overall, this work has required more than 3 years of research in collaboration with STMicroelectronics.

References

[1] U. Abelein, H. Lochner, D. Hahn, and S. Straube, "Complexity, quality and robustness - the challenges of tomorrow's automotive electronics," *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2012, pp. 870-871.

[2] U. Backhausen *et al*., "Robustness in automotive electronics: An industrial overview of major concerns," *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, Thessaloniki, 2017, pp. 157-162.

[3] Infineon, "*Automotive application guide,*" sensors microcontroller and power devices used an automotive field, 2019, testing on https://www.infineon.com/dgdl/Infineon-Automotive-Application-Guide-2019-ABR-v01_00-EN.pdf?fileId=5546d462584d1d4a015891808e617573

[4] S. Jeon, J. Cho, Y. Jung, S. Park, and T. Han, "Automotive hardware development according to ISO 26262," *13th International Conference on Advanced Communication Technology (ICACT2011)*, Seoul, 2011, pp. 588-592.

[5] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," IEEE Transactions on Computers, vol. 65, no. 3, pp. 744–754, March 2016.

[6] H. Gall, "Functional safety IEC 61508 / IEC 61511 the impact to certification and the user," *2008 IEEE/ACS International Conference on Computer Systems and Applications*, Doha, 2008, pp. 1027-1031.

[7] ISO26262, "*Road vehicles – functional safety*," 2011 https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-1:v1:en

[8] https://www.synopsys.com/support/training/signoff/tmax1-fcd.html

[9] https://www.synopsys.com/verification/simulation/z01x-functional-safety.html

[10] E. Fujiwara, "Code Design for Dependable Systems: Theory and Practical Application," New York, NY, USA, Wiley, 2006

[11] J. Teifel, "Self-Voting Dual-Modular-Redundancy Circuits for Single-Event-Transient Mitigation," in *IEEE Transactions on Nuclear Science*, vol. 55, no. 6, pp. 3435-3439, Dec. 2008.

[12] J. Yeh, K. Cheng, Y. Chou and C. Wu, "Flash Memory Testing and Built-In Self-Diagnosis with March-Like Test Algorithms," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1101-1113, June 2007.

[13] C. L. Chen and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review," in *IBM Journal of Research and Development*, vol. 28, no. 2, pp. 124-134, March 1984.

[14] Thatte and Abraham, "Test generation for microprocessors," IEEE Transactions on Computers, vol. C-29, no. 6, pp. 429–441, June 1980.

[15] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, and Y. Zorian, "Deterministic software-based self-testing of embedded processor cores," in Proceedings of the Conference on Design, Automation and Test in Europe, ser. DATE '01. Piscataway, NJ, USA: IEEE Press, 2001, pp. 92–96.

[16] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," IEEE Design Test of Computers, vol. 27, no. 3, pp. 4–19, May 2010.

[17] G. Theodorou, N. Kranitis, A. Paschalis and D. Gizopoulos, "Software-Based Self-Test for Small Caches in Microprocessors," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 12, pp. 1991-2004, Dec. 2014, doi: 10.1109/TCAD.2014.2363387.

[18] https://www.st.com/en/embedded-software/stm32-classb-spl.html and https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/group0/e6/0c/d5/34/97/d2/45/6f/China_ST_MCU_Technical_Day_3_SPC5_Ecosystem_Introduction/files/China_ST_MCU_Technical_Day_3_SPC5_Ecosystem_Introduction.pdf/jcr:content/translations/en.China_ST_MCU_Technical_Day_3_SPC5_Ecosystem_Introduction.pdf

[19] https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/

[20] http://www.cypress.com/file/249196/download

[21] https://www.renesas.com/us/en/products/synergy/software/add-ons/s7g2-iec60730-self-test-library.html

[22] http://ww1.microchip.com/downloads/en/DeviceDoc/52076a.pdf

[23] https://www.arm.com/products/development-tools/embedded-and-software/software-test-libraries

[24] M. A. Skitsas, C. A. Nicopoulos, and M. K. Michael, "DaemonGuard: O/S-assisted selective software-based Self-Testing for multi-core

systems," *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS),* New York City, NY, 2013, pp. 45-51.

[25] E. Sanchez, "Increasing reliability of safety critical applications through functional based solutions," *2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*, Taormina, 2018, pp. 1-1.

[26] P. Bernardi, S. De Luca, D. Piumatti, S. Regis, E. Sanchez, and A. Sansonetti, "On the in-field testing of spare modules in automotive microprocessors," *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Abu Dhabi, 2017, pp. 1-6.

[27] P. Bernardi, R. Cantoro, S. De Luca, E. Sanchez, A. Sansonetti, and G. Squillero, "Software-Based Self-Test Techniques for Dual-Issue Embedded Processors," in *IEEE Transactions on Emerging Topics in Computing*.

[28] D. Changdao, M. Graziano, E. Sanchez, M. Sonza Reorda, M. Zamboni and N. Zhifan, "On the functional test of the BTB logic in pipelined and superscalar processors," *2013 14th Latin American Test Workshop - LATW*, Cordoba, 2013, pp. 1-6.

[29] P. Bernardi *et al*., "On the in-field functional testing of decode units in pipelined RISC processors," *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Amsterdam, 2014, pp. 299-304.

[30] R. Cantoro, D. Piumatti, P. Bernardi, S. De Luca, and A. Sansonetti, "In-field functional test programs development flow for embedded FPUs," *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Storrs, CT, 2016, pp. 107-110.

[31] F. Corno, E. Sanchez, and G. Squillero, "Evolving assembly programs: how games help microprocessor validation," in *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 695-706, Dec. 2005.

[32] P. Beranrdi, R. Cantoro, A. Floridia, D. Piumatti, C. Pogonea, A. Ruospo, E. Sanchez, S. De Luca, A. Sansonetti, "*Non-Intrusive Self-Test Library for Automotive Critical Applications: Constraints and Solutions,*" 2019 IEEE Design, Automation and Test in Europe (DATE), Florence, Italy, 25-29 March 2019

[33] "*Power PC Embedded Application Binary Interface (EABI): 32-Bit Implementation*," https://www.nxp.com/docs/en/application-note/PPCEABI.pdf

[34] A. Floridia, E. Sanchez and M. Sonza Reorda, "Fault Grading Techniques of Software Test Libraries for Safety-Critical Applications," in *IEEE Access*, vol. 7, pp. 63578-63587, 2019.

[35] P. Bernardi, M. Grosso, E. Sanchez and O. Ballan, "Fault grading of software-based self-test procedures for dependable automotive applications," *2011 Design, Automation & Test in Europe*, Grenoble, 2011, pp. 1-2.

[36] P. Bernardi, D. Piumatti and E. Sanchez, "Facilitating Fault-Simulation Comprehension through a Fault-Lists Analysis Tool," *2019 IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS)*, Armenia, Colombia, 2019, pp. 77-80.

[37] A. Floridia and E. Sanchez, "Hybrid on-line self-test strategy for dualcore lockstep processors," in 2018 *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology* Systems (DFT), Oct 2018, pp. 1–6.

[38] P. Bernardi, L. M. Ciganda, E. Sanchez, and M. Sonza Reorda, "Mihst: A hardware technique for embedded microprocessor functional on-line self-test," *IEEE Transactions on Computers*, vol. 63, no. 11, pp. 2760–2771, Nov 2014.

[39] P. Bernardi, R. Cantoro, L. Gianotto, M. Restifo, E. Sanchez, F. Venini, and D. Appello, "A dma and cache-based stress schema for burn-in of automotive microcontroller," in 2017 *18th IEEE Latin American Test Symposium (LATS),* March 2017, pp. 1–6.

[40] T. F. Hsieh, J. F. Li, K. T. Wu, J. S. Lai, C. Y. Lo, D. M. Kwai, and Y. F. Chou, "Software-hardware-cooperated built-in self-test scheme for channel-based drams," in 2017 *International Test Conference in Asia (ITC-Asia),* Sept 2017, pp. 107–111.

[41] A. Floridia, G. Mongano, D. Piumatti, E. Sanchez, "Hybrid on-line self-test architecture for computational units on embedded processor cores," in 2019 *International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS),* April 2019

[42] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis and D. Gizopoulos, "Hybrid-SBST Methodology for Efficient Testing of Processor Cores," in IEEE Design & Test of Computers, vol. 25, no. 1, pp. 64-75, Jan.-Feb. 2008, doi: 10.1109/MDT.2008.15.

[43] "Specification of CommunicationAUTOSAR CPRelease 4.3.1," https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_COM.pdf

[44] "Embedded systems," Carlo Brandolese, William Fornaciari, Pearson Prentice Hall, 2007, ISBN 9788871923420

[45] A. Salem, "Formal verification of digital circuits," *4th IEEE International Workshop on System-on-Chip for Real-Time Applications*, Banff, Alta., Canada, 2004, pp. 15-15.

[46] "Specification of I/O Hardware Abstraction AUTOSAR CP Release 4.3.0," https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_IOHardwareAbstraction.pdf

[47] "STMicroelectronics Automotive MCU Technical Day," 2017, https://www.stmicroelectronics.com.cn/content/ccc/resource/corporate/company/divisional_presentation/group0/25/a7/3b/ad/c4/dd/49/7b/13_AUTOSAR_Solution_Introduction_ST_MCU_KPIT/files/13_AUTOSAR_Solution_Introduction_ST_MCU_KPIT.pdf/_jcr_content/translations/en.13_AUTOSAR_Solution_Introduction_ST_MCU_KPIT.pdf

[48] "SPC5 MCAL overview," ZHANG Livia, STMicroelectronics, 2017, https://www.st.com/content/ccc/resource/corporate/company/divisional_presentation/group0/e7/97/fc/c6/d6/68/46/ce/8_SPC5_MCAL_overview_Zhang_Livia/files/8_SPC5_MCAL_overview_Zhang_Livia.pdf/_jcr_content/translations/en.8_SPC5_MCAL_overview_Zhang_Livia.pdf

[49] "*Intel Unveils Sunny Cove, Gen11 Graphics, Xe Discrete GPU, 3D Stacking*," Paul Alcorn December 12, 2018. Web article available on https://www.tomshardware.com/reviews/intel-sunny-cove-gen11-xe-gpu-foveros,5932-4.html

[50] Intel public financial reports information available on https://www.intc.com/investor-relations/financials-and-filings/earnings-results/default.aspx

[51] "SPC5 32-bit microcontroller Series featuring Power Architecture," January 2016, document available on https://www.st.com/content/ccc/resource/sales_and_marketing/presentation/product_presentation/81/61/89/8b/77/1b/42/5f/SPC5_Family_Overview.pdf/files/SPC5_Family_Overview.pdf/jcr:content/translations/en.SPC5_Family_Overview.pdf

[52] RM0391 STMicroelectronics Reference manual of SPC58xEx/SPC58xGx 32-bit Power Architecture microcontroller for automotive ASILD applications, STMicroelectronics, August 2018, available on https://www.st.com/content/ccc/resource/technical/document/reference_manual/b9/33/31/8b/31/d0/4f/f6/DM00148989.pdf/files/DM00148989.pdf/jcr:content/translations/en.DM00148989.pdf

[53] UM0438 User manual Variable-Length Encoding (VLE) extension programming interface manual, STMicroelectronics, July 2007, available on https://www.st.com/content/ccc/resource/technical/document/user_manual/ac/f2/bf/01/73/d8/48/e0/CD00161395.pdf/files/CD00161395.pdf/jcr:content/translations/en.CD00161395.pdf

[54] RM0004 Programmer's reference manual for Book E processors, STMicroelectronics, May 2015, available on https://www.st.com/content/ccc/resource/technical/document/reference_manual/8b/6f/4e/d6/72/82/45/78/CD00164807.pdf/files/CD00164807.pdf/jcr:content/translations/en.CD00164807.pdf

[55] Lightweight Signal Processing APU (LSP APU) Reference Manual, Document Number: LSPAPURM Rev. 3, 12/2012, available on http://cache.freescale.com/files/microcontrollers/doc/ref_manual/LSPAPURM.pdf

[56] "SPC582B60x, SPC582B54x, SPC582B50x 32-bit Power Architecture microcontroller for automotive ASIL-B applications," Datasheet, available on https://www.st.com/en/automotive-microcontrollers/spc582b60e1.html#resource

[57] P. Bernardi *et al*., "Software-based self-test techniques of computational modules in dual issue embedded processors," *2015 20th IEEE European Test Symposium (ETS)*, Cluj-Napoca, 2015, pp. 1-2.

[58] Kyung-JungLee, Young-HunKi, and Hyun-SikAhn, "*Automotive ECU Design with Functional Safety for Electro-Mechanical Actuator Systems,*" World Academy of Science, Engineering and TechnologyInternational Journal of Computer and Systems Engineering Vol:7, No:7, 2013, available on https://waset.org/publications/16464/automotive-ecu-design-with-functional-safety-for-electro-mechanical-actuator-systems

[59] Burim ALIU, "Design and Implementation of a Self-test Concept for an Industrial Multi-core Microcontroller," Institut f¨ur Technische Informatik Technische Universit¨at Graz, May 2012, available on:

https://diglib.tugraz.at/download.php?id=576a76434c02d&location=browse

[60] A. Apostolakis, M. Psarakis, D. Gizopoulos and A. Paschalis, "A Functional Self-Test Approach for Peripheral Cores in Processor-Based SoCs," *13th IEEE International On-Line Testing Symposium (IOLTS 2007)*, Crete, 2007, pp. 271-276

[61] M. Grosso, W. J. H. Perez, D. Ravotto, E. Sanchez, M. S. Reorda and J. V. Medina, "A software-based self-test methodology for system peripherals," 2010 15th IEEE European Test Symposium, Praha, 2010, pp. 195-200

[62] A. Floridia, D. Piumatti, E. Sanchez, S. De Luca, and A. Sansonetti, "Parallel software-based self-test suite for multi-core system-on-chip: Migration from single-core to multi-core automotive microcontrollers," 2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), Taormina, 2018, pp. 1-6.

[63] A. Floridia et al., "Deterministic Cache-based Execution of On-line Self-Test Routines in Multi-core Automotive System-on-Chips," 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2020, pp. 1235-1240

[64] SPC58xNx 32-bit Power Architecture® microcontroller for automotive ASILD applications (Reference manual RM0421, DocID028528), STMicroelectronics, available on https://www.st.com

[65] Z. Jianwu, S. Yibing and Li Yanjun, "Aliasing Probability for Single Input Linear Feedback Signature Registers," *2007 8th International Conference on Electronic Measurement and Instruments*, Xi'an, 2007, pp. 3-995-3-999

[66] I. Voyiatzis, "On reducing aliasing in accumulator-based compaction," 2008 3rd International Conference on Design and Technology of Integrated Systems in Nanoscale Era, Tozeur, 2008, pp. 1-12

[67] I. Voyiatzis, "Aliasing Reduction in Accumulator-Based Response Verification," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 11, pp. 1746-1750, Nov. 2014

[68] S. R. Aruna and K. S. Neelukumari, "MISR architectures to remove unknown values in output response compaction," International Conference on Information Communication and Embedded Systems (ICICES2014), Chennai, 2014, pp. 1-4