On-line Software-Based Self-Test in Automotive Electronics: Problems and Solutions

P. Bernardi, M. De Carvalho, M. Grosso, J. Lagos, E. Sanchez
Dipartimento di Automatica e Informatica
Politecnico di Torino, Torino, Italy
{paolo.bernardi, mauricio.decarvalho, michelangelo.grosso, jorge.lagos, ernesto.sanchez}@polito.it

O. Ballan
STMicroelectronics
Agrate Brianza (MI), Italy
oscar.ballan@st.com

Abstract- **Today's increasing demand of electronic systems in the automotive field, enabling car makers to maintain a competitive edge in the market, raises new issues regarding safety and dependability. Emerging standards, such as ISO 26262, define requirements for these systems on final test as well as on on-line fault detection policies. This paper describes an integrated framework for the generation and the application of on-line Software-Based Self-Test (SBST) programs, currently experimented within STMicroelectronics.**

I. Introduction

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

The use of such devices in safety and mission-critical applications raises the need for total dependability. This requirement translates in a series of system audit processes that need to be applied throughout the product lifecycle. Some of these processes are common in today's industrial design and manufacturing flows, and include design verification and validation, performed from the early phases of product development, as well as various test operations during and at the end of manufacturing and assembly steps. More often, additional test operations need to be applied also during the product mission life, and may include periodic on-line testing and/or concurrent error detection [1]. The reliability requirements need to be met by trading off fault/error coverage capabilities with admissible implementation costs of the selected solutions.

Within the scope of microprocessor-based integrated systems, the on-line SBST approach has been addressed for a long time by the research community [2]. Compared to hardware-based test solutions, such as Built-In Self-Test (BIST) it presents many advantages, such as the possibility of autonomously testing both the microprocessor and the controllable peripherals in normal mode of operation, with no hardware modifications needed, and at-speed test application. Conversely, SBST methodologies raise some issues that have been limiting their application in industry throughout the years: those issues regard writing efficient and effective test programs and devising suitable methodologies for test application.

Concerning in-field test application, a possible solution [3] relies in periodic test application during the system idle time (Fig. 1). In simple words, the microprocessor is periodically (e.g., after each boot sequence at vehicle key-on) forced to execute a self-test code able to detect the possible occurrence of permanent faults in the processor core itself and the peripherals connected to it.

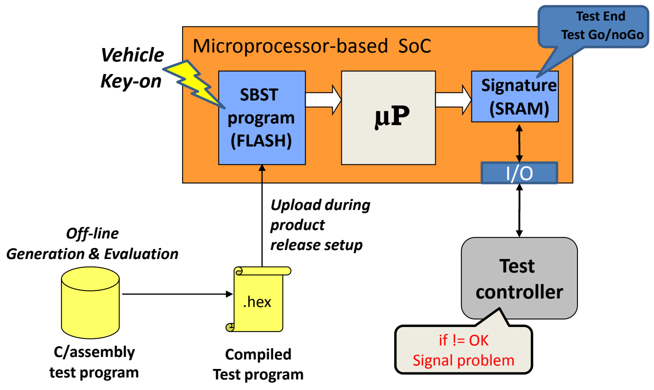

Figure 1: Conceptual representation of the in-field self-test procedure execution.

In the aforementioned approach, the SBST procedure is uploaded in a non-volatile (Flash) memory during the final device test; such procedure writes the self-test results in an available memory space, and then these are read out through a suitable serial port by an external Test Controller that inhibits the vehicle functional operations in case a faulty behavior is identified.

As far as it concerns test program development, many approaches can be found in the literature [4], employing manual or automated approaches, which are able to target different processor architectures and fault models. However, setting up an efficient SBST program generation framework within a consolidated industrial system development flow involves a number of issues to be solved regarding the development and grading of the test programs.

This paper presents an approach currently experimented by STMicrolectronics for the development and evaluation of SBST programs for microprocessors in safety-critical embedded systems. The pursued goal is to satisfy the reliability requirements given by emerging standards such as ISO/FDIS 26262 [5], which mandate a constant monitoring for the possible occurrence of permanent faults in the circuit (Table I). The actual test program generation is carried out employing manual and automated techniques, and an integrated framework has been setup for circuit analysis and partitioning and for fault grading of the generated programs.

TABLE I: SINGLE POINT FAULTS METRIC AND LATENT FAULTS METRIC TARGET VALUES [5].

	ASIL B	ASIL C	ASIL D
Single point faults metric	> 90%	> 97%	> 99%
Latent faults metric	> 60%	> 80%	> 90%

II. THE PROPOSED SBST PROGRAM GENERATION FRAMEWORK

Fig. 2 presents a comprehensive view of the proposed test generation framework. The goal of this work is to obtain high fault coverage while satisfying constraints in terms of test code memory occupation and test duration.

The first phase of the flow is *circuit partitioning*: this step is based on the analysis of the hierarchical RTL description in order to identify the main microprocessor modules, for which the test program generation development will be carried out following a *divide et impera* strategy so as to reduce the problem complexity.

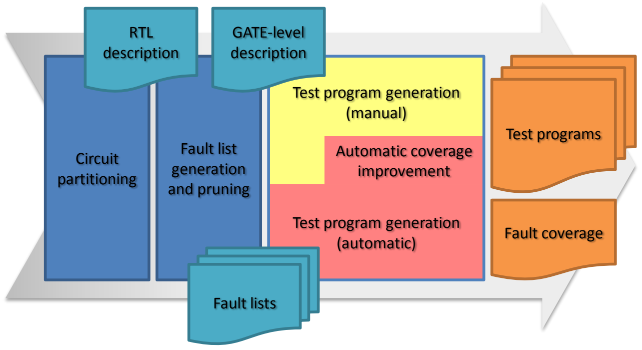
RTL description
GATE-level description
Circuit partitioning
Fault list generation and pruning
Test program generation (manual)
Automatic coverage improvement
Test program generation (automatic)
Test programs
Fault coverage
Fault lists

Fig. 2: Conceptual view of the proposed integrated test generation framework

The gate-level description is then taken into account in the *fault list generation and pruning* phase, to generate the list of faults according to the previously identified module subdivision. In addition, untestable faults are removed to reduce the fault list size and to enable more accurate fault coverage computation in the following phase. Among the untestable faults that can be identified in this step there are the ones related to scan chains, which are usually implemented in the design and used for end-of-production testing, but become useless at mission time.

The following phase is the actual *test program generation* process. In the proposed framework, different strategies are employed to obtain effective test program, depending on the addressed microprocessor module. The different techniques are supported by a suitably developed fault grading environment [6], which provides a quantitative estimation of the effectiveness of the generated programs and useful information for their improvement. Further details about the fault grading process are given in the next subsection.

The *manual test program generation* process resorts to state-of-the-art methodologies that usually guarantee satisfying coverage results for the most common modules, such as the Arithmetic-Logic Unit (ALU) and the register file. Additionally, through manual generation of test programs, it is possible to explore particular processor conditions or activate specific corner cases, such as the operating modes dealing with the interrupt mechanism. An iteration flow guided by the outcome of the grading process is usually employed.

In parallel to the manual approach, an *automated test program generation* methodology is used, based on an evolutionary optimization engine [7]. Fig. 3 shows the employed loop-based flow, presenting the three main blocks involved in an automatic run: an evolutionary engine, a constraint library, and an evaluator external to the evolutionary core (in this case, the fault grading environment). The constraint library stores suitable information about the microprocessor assembly language. The evolutionary engine, on the other hand, generates an initial set of random programs, or individuals, exploiting the information provided by the constraint library. Then, these individuals are cultivated following the Darwinian concepts of evolution, by applying mutations to the various individuals. The evaluation of the test programs is carried out through fault simulation, which evaluates different test metrics (fault coverage, memory occupation of the test code, execution runtime) and provides a *fitness* value which is used to guide the optimization process.

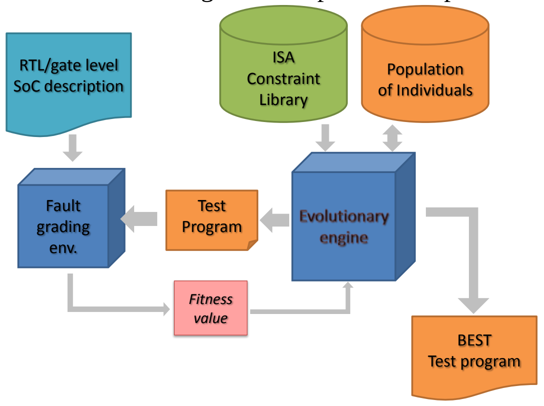

Fig. 3: Feedback-based automated program generation flow.

The described evolutionary flow is also employed to increase the fault coverage figures of test programs for microprocessor modules initially tackled through manual approaches, in the so-called *automatic coverage improvement* phase, when the algorithms employed in the manual flow are unable to provide satisfying coverage values. This is usually due to the peculiar architectures of the addressed devices or to specific synthesis choices.

The obtained set of test programs still needs a manual refinement phase, to merge the obtained programs so as to build a comprehensive test procedure and, if needed, to cover additional corner cases or optimize memory occupation and test execution runtime. The obtained test procedure is finally prepared to be stored in the system code memory and run at the selected circumstances.

A. Fault grading environment

Fault grading is a extremely critical step of the test program generation flow, for different reasons. First of all, the output fault coverage figures need to provide a realistic information about the ability of the evaluated program in mission-like conditions. As an example, excessively optimistic results may lead to assume some faults covered by a test program, while in the field their effect may be masked during test application due to the lack of observability; conversely, overly pessimistic figures may uselessly make test program generation harder. In addition, since during the test program development a great number of programs usually need to be evaluated, the fault grading environment needs to be optimized so as to reduce the needed computational resources and time.

Fig. 4 presents the adopted fault grading environment[6] . The main phases are

- *test program preparation*, where the developed high-level code is compiled and translated in a simulation-compliant format, and the code memory occupation is evaluated;
- *test program simulation*, during which the programs are run in the system using a suitable testbench, and a trace dump of the input/output signals of the addressed microprocessor module is produced. In addition, test application time is computed;
- *test program grading*, where the actual fault simulation is performed on the selected module, using as input the previously produced signal dump.

The most critical details to be taken into account are highlighted in red and blue in the figure and concern the selection of the circuit portion to be fault simulated and the signals to be observed to distinguish among correct and corrupted behavior. *Hierarchy selection* reflects the choices taken during circuit partitioning. A smaller circuit portion makes the fault simulation process faster, however, its ports may not be directly observable during test, and a higher hierarchy level may need to be employed in order to achieve fair fault coverage figures. *Observability selection* is another crucial point in the entire flow, since an inaccurate choice may lead to imbalanced measurements. If an internal hierarchy level has been selected during the simulation phases, all signals involved in the transport of observable fault effects outside the selected circuit zone have to be considered for observation, while the others must be left out. Furthermore, these signals have to be observed only during time instants when significant information is transmitted.

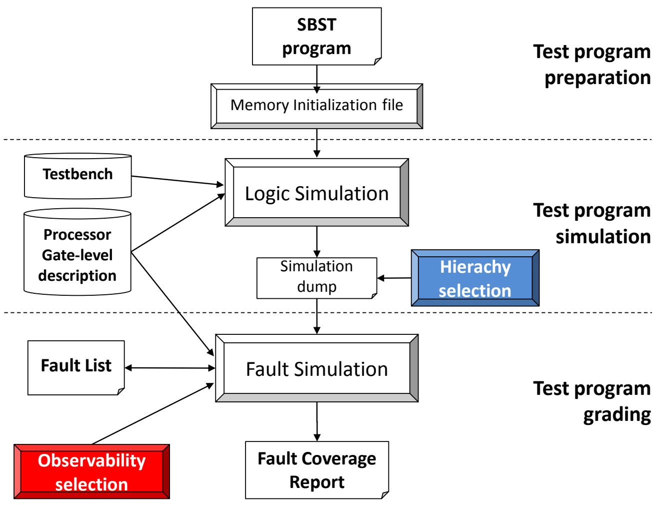

Fig. 4: Fault grading environment.

III. CASE STUDY

The proposed test program generation framework is currently used to develop SBST program suites in STMicroelectronics microprocessor-based systems for the automotive field.

The proposed test program generation framework has been built on the following tools:

- Cadence Incisive suite for fault-free simulation
- Synopsys TetraMAX for fault simulation
- µGP3[3] [7] as the evolutionary engine
- Ad-hoc programs, suitable developed to support the flow (fault list pruning, data format conversion, etc.)

The experiments are run on a workstation featuring two Intel Xeon E5450 processors (8 processing cores running at 3 GHz) and 8 GB RAM.

As a case study, a SoC including a 32-bit pipelined MCU based on the Power Architecture™ is used for illustrating the proposed flow. It features a 576 KB Flash memory and 40 KB RAM, and is used for automotive chassis and safety applications, such as in airbag, ABS and EPS controllers (Fig. 5). The focus of the program generation process was put on the SoC module called *platform* which includes the most computation-intensive and programmable components of the SoC. The addressed circuit modules include about 150K gates and 400K stuck-at faults. Figure 4 shows the SoC architecture and illustrates the hierarchical level chosen for fault simulation (in blue), the signals selected for observation (in red) and areas where untestable faults have been pruned (in green).

The untestable faults removed from the fault lists include the embedded software debug infrastructure and the faults belonging exclusively to scan chain devoted components. The pruning operations were performed relying on ad-hoc tools working on the gate-level netlist in Verilog format.

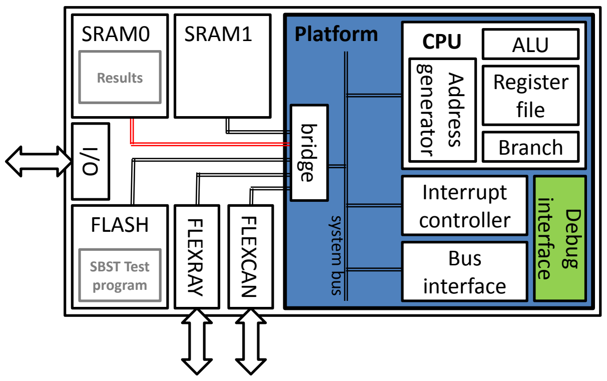

Fig. 5: SoC architecture with hierarchy selection (in blue), observability selection (in red) and one of the untestable fault pruning zones (in green).

Table II reports some preliminary figures regarding test program generation for some datapath components, obtained employing the automated and the manual test program generation flows. For the presented results, no methodologies were used to reduce test execution runtime; further program refinement is currently ongoing. Concerning the adder module, the table presents coverage values for a preliminary manually generated program and for the following automatic program improvement, which yielded a relevant coverage increment.

Fig. 5 illustrates the coverage evolution in time during the automatic improvement of test program for the adder module, starting from the coverage obtained with the manual flow.

IV. Conclusions

Periodic on-line software-based self-test is a viable solution for guaranteeing system dependability in safety-critical systems. We presented a test program generation framework currently under experimentation within STMicroelectronics. Works are ongoing to optimize and increase the automation of the developed flows. Additionally, we are working to integrate the evaluation of RTL code coverage metrics computed during simulation in order to accelerate test program generation in both manual and automated flows.

References

[1] M. Nicolaidis, Y. Zorian, "On-line testing for VLSI—a compendium of approaches", Springer Journal of Electronic Testing: Theory & Applications, Vol. 12, N. 1-2, 1998, pp. 7-20

[2] A. Paschalis, D. Gizopoulos, "Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors", IEEE Trans. on CAD, Vol. 24, n. 1, Jan. 2005, pp. 88-99

[3] O. Ballan, P. Bernardi, G. Fontana, M. Grosso, E. Sanchez, "Fault Grading of Software-Based Self-Test Procedures for Dependable Automotive Applications", IEEE Design, Automation and Test in Europe Conference, 2011

[4] M. Psarakis, D. Gizopoulos, E. Sanchez, M. Sonza Reorda, "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, Vol. 27, n. 3, 2010, pp. 4-19

[5] ISO/FDIS 26262 "Road vehicles – Functional Safety"

[6] O. Ballan, P. Bernardi, G. Fontana, M. Grosso, E. Sanchez, "A Fault Grading Methodology for Software-Based Self-Test Programs in Systems-on-Chip", IEEE International Workshop on Microprocessor Test and Verification, 2010

[7] E. Sanchez, M. Schillaci, G. Squillero, "Evolutionary Optimization: the µGP toolkit", Springer, 2011

[8] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis, "Software-Based Self-Testing of Embedded Processors", IEEE Trans. on Computers, Vol. 54, n. 4, Apr. 2005, pp. 461-475

[9] M. Psarakis, D. Gizopoulos, A. Paschalis, "Built-In Sequential Fault Self-Testing of Array Multipliers", IEEE Trans. on CAD, Vol. 24, n. 3, Mar. 2005, pp. 449-460

Table II: Results of preliminary SBST program generation on some microprocessor modules.

Module	**Generation methodology**	**Stuck-at faults [#]**	**Stuck-at fault coverage [%]**	**Development time**	**Clock cycles**	**Memory [Byte]**
Register file	manual [8]	61,327	58.27	3 days	747	1500
Adder	manual [8]	4,188	56.67	3 days	9,013	160
	aut. improvement [7]		91.79 (incr.)	10 days	1,384	838
Multiplier	manual [9]	31,809	95.25	3 days	376K	331

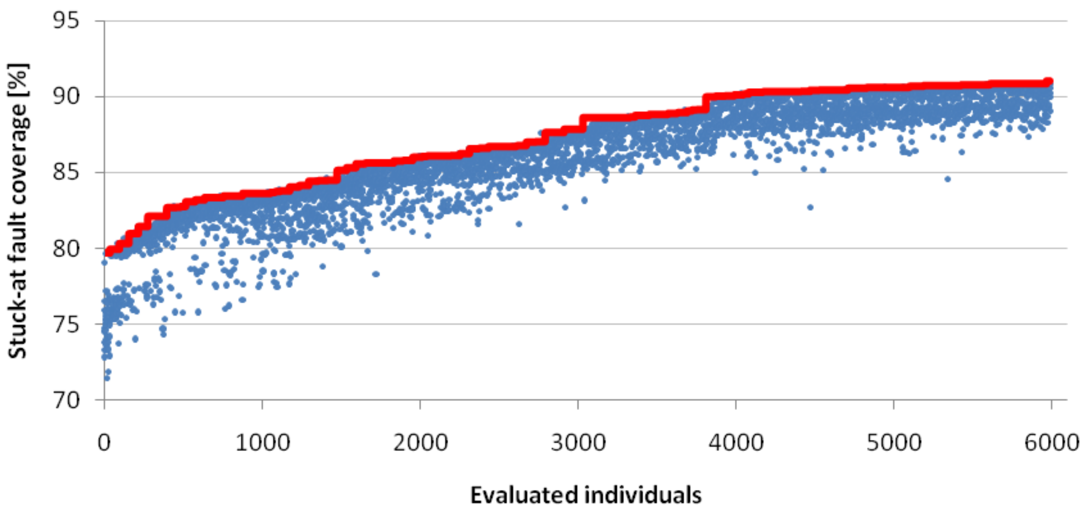

Fig. 5: Evolution of test program fault coverage during automated program generation for the adder module; each blue point represents an evaluated individual, while the red line traces the generation best individuals.