

An analysis of test solutions for COTS-based systems in space applications

R. Cantoro, S. Carbonara, A. Florida, E. Sanchez, M. Sonza Reorda
Politecnico di Torino, Torino, Italy

Jan-Gerd Mess
DLR, Bremen, Germany

Abstract—One of the current trends in space electronics is towards considering the adoption of COTS components, mainly to widen the spectrum of available products. When substituting space-qualified components with COTS ones a major challenge lies in guaranteeing the same level of reliability. To achieve this goal, a mix of different solutions can be considered, including effective test techniques, able to guarantee a high level of permanent fault coverage while matching several constraints in terms of system accessibility and hardware complexity. In this paper, we describe an approach based on Software-based Self-test, which is currently being adopted within the MaMMoTH-Up project, targeting the development of an innovative COTS-based system to be used on the Ariane5 launcher. The approach aims at testing the OR1200 processor adopted in the system, combined with new and effective techniques for identifying the safe faults. Results also include a comparison between functional and structural test approaches.

I. INTRODUCTION

Space applications are known to be extremely challenging from a reliability point of view, since they are supposed to work in a harsh environment (not only in terms of radiation but also from the point of view of stresses coming from extreme temperature, pressure, vibration, etc.) with strong requirements in terms of reliability. In order to reduce cost and especially to increase device availability, there is a trend towards the adoption of Commercial Off-The-Shelf (COTS) components instead of the space qualified ones. Obviously, this trend requires evaluating the costs and efforts for guaranteeing that the resulting reliability still reaches the target threshold [2]. A special niche within the general domain of space applications relates to launchers. In this case, the mission time is more reduced, while the radiation environment corresponds to all the layers from ground up to the geostationary orbit (GEO). The MaMMoTH-Up project [3], funded by the European Commission within the frame of the Horizon 2020 research and innovation programme, aims at developing and evaluating a COTS-based system to be used in the telemetry unit of the Ariane5 (A5) launcher. More in details, the MaMMoTH-Up system is composed of several boards targeting data acquisition and processing, power management, and data transmission. All these boards use COTS components, including a flash-based FPGA where several IPs are mapped, including an OpenRISC1200 (OR1200) processor [4] whose design has been properly modified to harden it with respect to radiation effects. The adoption of such processor allows the MaMMoTH-Up system to perform significantly more powerful functions than the system it is going to substitute, e.g., in terms of data analysis and compression. In order to match the strict reliability targets of A5, the MaMMoTH-Up system must be protected not only from the radiation effects, which are mainly responsible for Latch-up and transient fault effects, but also from

possible permanent faults arising during both the manufacturing process and the following system life. To target permanent faults several test steps have been identified, which are performed during and at the end of the manufacturing process, at the end of the assembly step, and after the system is mounted in the final position. Some test is also performed during the mission. The fault coverage which can be achieved by these test steps is important, since it directly impacts the achieved reliability level. To estimate the Fault Rate of the different components, we followed the FIDES guidelines [5], taking into account the stress conditions which are applied to the system before and during the mission. The Failure Rate is then derived by applying an FMECA (Failure Mode, Effects, and Criticality Analysis) procedure [11] which identifies the fault effects (and their criticality) and takes into account the timing and effectiveness (i.e., the fault coverage) of the different test steps. Remarkably, some of them have to be performed while the system is already mounted in its final position. Hence, they must basically correspond to a self-test, during which some command is sent to the system, the system performs a test of the hardware, and then results are sent outside. In the previous versions of the target system, which was based on much simpler space qualified hardware, a *functional test* was used for this purpose, where the system was asked to perform some basic operations, and a check on the computed results was sufficient to identify possible faults. Due to the much higher complexity of the MaMMoTH-Up system this approach can hardly guarantee the achievement of the required fault coverage, especially on the OR1200 core. Hence, a *structural approach* has been devised, based on a set of self-test procedures in charge of checking the possible presence of permanent faults affecting the processor core. The key difference between the two approaches lies in the fact that the functional one checks whether the system is able to deliver the expected functions, while the structural one identifies first some fault model related to the implementation of the underlying circuit, and then tries to detect the resulting faults. A major advantage of the structural approach clearly lies in the fact that the adopted fault coverage metric can be more deterministically and quantitatively evaluated than for the functional approach. Moreover, while for simple systems the functional approach (if suitably implemented) can achieve a sufficient testing quality, for more complex modules (such as the OR1200 one) the same is not true, as we will experimentally show in the paper. When dealing with the OR1200 processor, the self-test procedures implementing the structural approach follow the Software-based Self-test (SBST) paradigm [6]. Their code is integrated in the application software and, when activated, forces the processor to execute a proper sequence of instructions. The produced results are compacted into a signature which is returned to the calling

program, which can thus check the possible presence of a fault by comparing it with the expected one.

The contribution of this paper lies first in describing a case of study (corresponding to the OR1200 core) where the characteristics of a functional and structural approach can be compared (not only in terms of achieved fault coverage, but also of memory footprint and duration). Secondly, it describes a scenario, where SBST can be effectively adopted, matching the several requirements of the qualification, acceptance and in-fly test of a space application. Finally, since the target system is expected to perform a well-defined set of functions and in a very specific configuration (e.g., in terms of memory address space), the FMECA is in charge of identifying which faults within the OR1200 core can produce any failure, and which faults will never be able to do so, e.g., because they relate to some hardware part which is not used by the application. While several techniques are available to automatically identify some categories of untestable faults, we focus here on those faults called *Safe faults*. These faults cannot produce any failure due to the specific (hardware or software) constraints the system matches during its normal operation. The paper shows that the number of safe faults is far from being negligible and uses an improved version of the method proposed in [7] to partly automate their identification. Due to the impact of the considered scenario, the fault coverage results reported in this paper are not directly comparable with those in [8], which focus on end-of-manufacturing test, although they refer to the same processor. The paper is organized as follows. Section II summarizes the main characteristics of the MaMMoTH-Up system, both in terms of underlying hardware and performed functions. Section III compares the functional and structural approaches, while Section IV focuses on the identification of safe faults. Section V finally draws some conclusions.

II. THE MAMMOTH-UP SYSTEM

A. General architecture and functions

The MaMMoTH-Up system shall provide an experiment and data acquisition opportunity on board the Ariane5 upper stage [3]. It is designed to offer the following functionalities:

1. Acquire measurement data
2. Configure and control the experiment
3. Provide a power supply
4. Perform self-testing and fault management.

To meet these functional requirements, a COTS-based system including one experiment controller (TCM-S), two computing nodes (OBC-S), two data acquisition boards (AQB) and a power supply unit (PSU) was developed. The system is housed in a foam-cushioned container to protect it from the harsh environment on board the launch vehicle. In order to collect sensor data and communicate with the Ariane5 upper stage, the system offers analogue acquisition channels for temperature, acceleration, vibration, shock and pressure sensors and a RS422 interface for data downlink. Synchronization with the launcher timeline and direct status reporting is done using three closed-current loops as inputs and eight discrete output pins. During the mission, the system steps through a number of different acquisition schemes according to the specific mission profile. An acquisition scheme determines which sensors are activated at which sampling rates up to 10 kHz. The data is collected and preprocessed by the computing nodes and then sent to the experiment controller using the internal SpaceWire bus. On the experiment controller, the data is analyzed, compressed and

stored on a flash-based mass memory before it is sent to the Ariane5 and downlinked using the launcher's telemetry chain. The complete data flow including its allocation to the different boards is depicted in Figure 1.

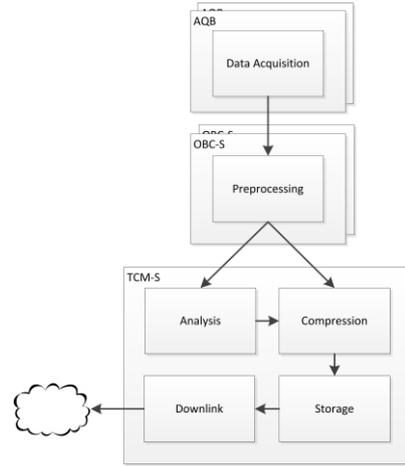


Figure 1 Data Flow

Each OBC-S board as well as the TCM-S board include a flash-based IGLOO2 FPGA. Each of these is holding an OR1200 soft core as well as accompanying IP cores, i.e. for SpaceWire communication amongst the boards. The required software images are kept in a two gigabyte NAND-flash memory that is implemented on each board. For data storage, the TCM-S is equipped with an additional sixteen gigabyte NAND-flash. The data acquisition is performed by a custom IP core that samples ADC channels and returns a block of samples to the software. Preprocessing, analysis and compression are then performed by software run by the OR1200 processors. The data compression algorithm consists of two steps whose load is divided between the OBC-S and the TCM-S. The OBC-S boards will perform a wavelet transform. The transformed data is then sent to the TCM-S. From the received wavelet transform, certain characteristics of the underlying data (e.g., value range and maximum gradient) are deduced. The transformed coefficients are then encoded into an embedded bitstream. According to the deduced characteristics of a given block, a certain number of bytes in the downstream are allocated for this bitstream. All other bits are cut to save downlink budget. The complete compression scheme is described in [9]. From a reliability point of view, although the OR1200 processors on the FPGAs and especially their memories and registers are hardened by duplication or triplication of some of the underlying flip-flops, there is no redundancy at the unit or system level. If the failure is not permanent, the system is able to recover by performing a software reset or power-cycle on the affected board. Should this not be successful because the failure proves to be permanent, the board has to be deactivated, inevitably resulting in a loss of the connected sensor channels. In this case, the MaMMoTH-Up system follows the concept of graceful degradation: although parts of the sensors cannot be acquired anymore, the remaining transfer budget can be reallocated to use it as efficiently as possible.

B. The OR1200 processor

The OR1200 is the only major RTL implementation of the OR1K architecture spec. The OR1200 is a 32-bit scalar RISC with Harvard micro-architecture and 5 stage integer pipeline. The OR1200 core is mainly intended for embedded, portable and networking applications. Fig. 2 shows its architecture.

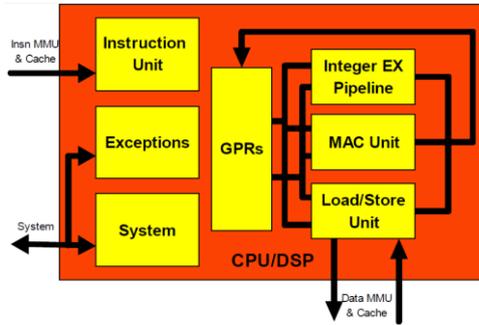


Figure 2. OR1200 CPU Architecture

III. COMPARING THE FUNCTIONAL AND STRUCTURAL APPROACHES

A. Background

In the frame of the actions to evaluate the reliability of the MaMMoTH-Up system and to guarantee that the target figures are matched, a key role is played by the test solutions adopted to identify possible permanent faults. These solutions are activated in different phases of the product life time, since the qualification step until the operational phase (i.e., during the launch). We underline that these test solutions should be usable and effective when applied in a scenario, where the target modules (e.g., the FPGA implementing the processor) have been already mounted on their boards, and each board has been included in the final box corresponding to the telemetry unit, which has been installed in its final location within the launcher. Hence, the whole test should be performed with very limited support from the outside, and should be minimally invasive with respect to the target system. In order to evaluate the effectiveness of these test solutions and to use meaningful fault coverage figures during the reliability evaluation process, a metric must be first identified. Traditionally, a functional metric is adopted. Since the early specification phases, the list of functions that the system must support is defined. For each of them, a *functional test* is then developed, aimed at verifying that the target function is correctly performed by the system. In this scenario a qualitative metric is adopted, which guarantees that the system is not affected by any fault if all the functional tests for all the functions are successful. When moving to more complex systems including COTS components, a different metric can be considered, which first identifies a structural fault model which is supposed to represent the possible permanent faults in the target device, and then computes the percentage of structural faults which are detected by the considered test solution. One of the goals of the MaMMoTH-Up project is to define new procedures for reliability evaluation, able to match the characteristics of COTS-based systems. Given their complexity, the project partners decided to assess the effectiveness of the functional approach when a structural fault model was adopted, at least for the most complex module, corresponding to the OR1200 CPU. Since the detailed information about the structure of the adopted FPGA were missed, we decided to perform such an assessment resorting to the popular stuck-at fault model, computing first the fault coverage achieved by the functional test when the CPU circuitry mapped on the FPGA was synthesized with a generic gate-level library. This approach is partly supported by the results of [20], showing that the stuck-at fault model, when applied to FPGAs, provides Fault Coverage results which are not far from those which can be obtained resorting to more accurate fault models, based on the knowledge of the internal implementation of the device (which is not available in our case).

Moreover, we developed a set of SBST test procedures targeting the stuck-at faults inside the OR1200 core. These procedures (that we cumulatively call *structural test* in the following) are integrated within the application software of the system and can be easily launched from the outside or by the system itself when required. Each of them returns a signature compacting the results produced by the test code, which can be compared with the expected one. A mismatch means that a permanent fault exists in the CPU core. In the following, we first report some information and figures (Table I) about the functional test and the structural one (based on SBST procedures) we developed. We will then report the experimental results aimed at comparing the effectiveness of the two test approaches (Table II).

B. The functional test

The functional test is composed of a compression algorithm that imposes a high workload on the arithmetic units of the processors. It is essential that the processor is fault-free, because even small changes in single bits of the output stream can result in a completely different set of data after decompression. Since it is impossible to predict the exact sensor readings, the processor cannot be checked using live data. Instead, precompiled blocks of sensor data together with expected values for the resulting transformation coefficients and bitstream are used. By comparing the output of the compression algorithm with the expected values, it is checked whether the calculations can be executed as planned. However, in case of an error, no diagnostic conclusions about the affected units within the processor can be drawn. The second line of Table I reports the size and duration of the functional test in terms of amount of memory to store the code and test time execution.

C. The structural test

The structural test is based on a suite of test procedures that target the different modules of the processor: program counter generator (*genpc*), instruction fetch (*if*), control unit (*ctrl*), register file (*rf*), operand muxes (*opmux*), arithmetic logic unit (*alu*), multiply and accumulate unit (*multmac*), load and store unit (*lsu*) and write back multiplexer (*wbmux*). Each test program executes a sequence of instructions aimed at stimulating as much as possible the target unit. At the end of the test, a signature is stored in memory: if the produced result is different than the expected one, it means that the CPU module is affected by a fault. All the test procedures have been written manually following the guidelines provided in [8]. In the following, we provide the most important characteristics of every one of the developed test programs.

The *genpc* and *if* modules are tested together using a single program. Any type of instruction from the Instruction Set must be tested. The program is written in such a way, that each type of instruction is followed by an unconditional jump to a procedure to the bottom of the code that updates the value of the signature and then it jumps back again to the top. In this way, the program counter adder inside the *genpc* is well tested, since it continuously jumps backward and forward, so performing additions and subtractions. In order to test the *ctrl* module, it is necessary to give it as inputs all the possible instructions from the Instruction Set: arithmetic, logic, branch, jump, compare, multiply, load and store, immediate or register-to-register. Since the *ctrl* module also generates signals to freeze some selected stages of the pipeline or to activate the forwarding when data hazards occur, it is important to include some instruction sequences with suitable data dependency in order to stimulate those signals. The values of the operands are not so important in

this case, so random values are chosen for the operations. The *rf* module is tested using register to register operations. Basically, the test consists in writing a value into a register and then reading it. The test is divided into four parts. In the first part of the test, the stack pointer and the link registers are tested. In the second part, the first half of the registers (r2-r15) is tested, assuming the other part is not faulty and using one among these registers to hold the signature. In the third part of the test, the second half is considered in turn, assuming the first part is not faulty. The values written in the registers are 0x55555555 and 0xAAAAAAAA. To protect the CPU core against temporary faults the register file has been duplicated and the first operand is read from one register, while the second operand is read from the second register; the write back operation updates both registers. Hence, it is necessary to perform each instruction twice, swapping the two operands, in order to read the values from both registers. The *opmux* module selects the operands for the execution units, choosing between values coming from the register file or from the various pipeline registers when forwarding is needed. The idea to test this module is to choose arithmetic, logic, load/store and multiply instructions in such a sequence that causes data dependencies in different stages of the pipeline. The *alu* module test addresses all the possible arithmetic/logic instructions of the instruction set. Some special values generated resorting to an Automatic Test Pattern Generation (ATPG) tool launched on the combinational part are chosen as operands to better test its functionalities and all the operations are performed choosing as operands all the possible combinations between the values above. The test of the *mult_mac* module depends significantly on the values chosen for the operands. Therefore, an ATPG tool has been used again to generate proper input values. The test program consists in a series of multiplications (also with immediate, signed and unsigned operands), multiply-accumulate and multiply-subtract instructions of the computed random operands. Division instructions also involve the *mult_mac* module to operate and it has also been tested. Since the *mac* instruction uses special purpose registers to accumulate, it is necessary to read the values written in these registers after each multiply-accumulate instruction. For testing the *lsu* module, all kinds of load and store instructions are considered: load/store byte or word, extended to zero or signed. The program is constituted of a sequence of instructions to write and read contiguous locations in memory; each block is composed of instructions performing the following three steps: a) Storing a value in a memory location, b) Reading the value from the same location, c) Updating the signature. The values chosen to be written in memory are random and the offset to be added to the base address is a large value (from 16,380 to 17,380). The *wbmux* module chooses the value to be written back into the register file, whether it comes from the memory system (for a load instruction) or from the execution units. Since this module basically corresponds to a mux, the program is very similar to that developed for the *opmux* module. Table I summarizes the characteristics of the Functional and Structural tests in terms of size and duration. For the Structural test, we detailed these figures for each test procedure.

D. Results

For the purpose of our experiments, we created a simulation setup where the OR1200 processor core lies in a system composed also of a 64 MB RAM, as in the MaMMoTH-Up OBC-S boards. The processor core has been synthesized targeting the NanGate 45nm Open Cell Library. The obtained netlist is used to perform the fault simulation experiments with a

commercial tool. Using this setup we evaluated the stuck-at Fault Coverage obtained by both the functional and the structural test described above. Results are reported in Table II for the whole CPU and for each component module. As the reader can notice, the Fault Coverage achieved by the Functional test is far lower than the one of the Structural test. This supports the claim that Functional test cannot be effectively used when complex COTS-based systems are used. It is also worth underlining that the comparison between the two tests provides very different results depending on the considered module. For some of them (e.g., *genpc*) the fault coverage achieved by the Functional test is slightly higher than by the Structural test. This is basically due to the fact that some modules can be tested in a good way by executing long programs, and the Functional test is much longer than the Structural one. However, for modules that include large combinational parts (e.g., *alu* and *mult-mac*) or require a specific sequence of operations to be tested (e.g., *rf*) the Structural approach is far more effective.

Table I. Characteristics of the test programs

	Size [Byte]	Duration [#clock cycles]
Functional test	17,360	379,815
Structural test	25,676	74,761
<i>genpc-if</i>	2,896	41,635
<i>ctrl</i>	980	980
<i>rf</i>	10,076	7,281
<i>opmux</i>	544	508
<i>alu</i>	3,184	10,497
<i>multmac</i>	2,996	9,962
<i>lsu</i>	4,244	3,224
<i>wbmux</i>	756	674

IV. SAFE FAULTS

A. Safe faults

We denote as *Safe Faults* those faults that can never produce a failure in the considered system¹. One of the goals of the FMECA is their identification, since they do not contribute to the Failure Rate, and should thus be removed from the Fault List to be considered when evaluating the Fault Coverage achieved by the test procedures. The ISO 26262 standard for automotive applications define them as “application dependent safe faults”. Clearly, Safe Faults include untestable faults. Hence, it can be useful to review the different categories included in the set of Safe Faults for a given system:

- *Structurally (or combinationally) untestable faults*, i.e., faults for which a test does not exist even if the combinational block where the fault is located is fully controllable and observable (e.g., via scan). Examples of faults belonging to this category include faults that cannot be tested due to some redundancy in the combinational logic. If a gate-level description of the device is available, an ATPG tool can identify these faults.
- *Sequentially untestable faults*, i.e., faults that do not belong to the previous group, but cannot be tested due to the sequential behavior of the circuit, for example, because the circuit cannot

¹ When performing FMECA, it is common to also distinguish between failures depending on their criticality, i.e., on how serious the effects of these failures are. Reliability figures typically depend only on *critical* safe faults. For the purpose of this paper we ignore any distinction within the set of safe faults.

reach any of the states required for their test. Several works proposed techniques to automatically identify these faults, either in a generic circuit [12][13][14][17][18] or specifically in a CPU [15].

- *On-line functionally untestable faults* [10], i.e., faults that do not belong to the previous groups, but cannot be tested in a functional manner (i.e., without resorting to Design for Testability) in the operational conditions the target device works in. On-line functionally untestable faults can be related for example to the specific memory configuration adopted by the system [16]. Several bits in the processor Program Counter or in the registers storing the addresses in the Load-Store Units become untestable if the memory area storing the code and the data is less than the maximum one.

Table II. Stuck-at (SA) fault coverage of functional and structural test on OR1200.

Module	Total SA faults	Functional test	Structural test
CPU	124,612	32.09 %	81.89 %
genpc	4,906	60.80 %	57.97 %
if	2,268	50.57 %	71.12 %
ctrl	4,320	71.53 %	80.25 %
rf	39,056	33.97 %	90.93 %
opmux	2,530	90.51 %	96.05 %
alu	14,532	46.04 %	78.50 %
mult_mac	39,398	13.91 %	95.77 %
sprs	5,522	8.31 %	37.61 %
lsu	2,708	67.61 %	65.99 %
wbmux	2,286	69.29 %	78.83 %
freeze	126	75.40 %	76.98 %
except	6,716	15.86 %	18.92 %
cfgr	232	0.00 %	0.00 %

Safe faults include and extend the previous categories. In the following we report some examples of safe faults:

- The debug circuitry possibly existing in a processor generates safe faults, since debug facilities are not used during the normal behavior, and several faults within it cannot impact the system behavior and produce any failure.
- Several faults in the Design for Testability hardware (e.g., the scan chains) used for end-of-manufacturing test also correspond to safe faults: for example, faults on the scan-in input of the scan flip-flops are safe faults.

In [7], we reported some results concerning the identification of safe faults in the openMSP430 processor. In that paper, we also considered those safe faults that cannot produce any failure, due to the specific application code executed by the CPU. As a simple example, if the system application only uses integer arithmetic, faults in the Floating Point Unit become untestable.

B. Safe faults identification

The typical approach for safe faults identification is based on manual analysis. In many project teams, the designers, test engineers, and reliability/functional safety experts systematically meet to categorize faults and (based on their effects) identify safe faults. Clearly, this process is extremely time consuming (and hence expensive), as well as prone to possible errors. For this reason, in [7] we recently proposed an approach, which aims at partly automating the safe faults identification process taking into account all the constraints coming from the application scenario, including the application software to be run by the CPU. Some preliminary results coming from the application of the same method to the OR1200 CPU have been reported in [19]. In this paper, we improve the procedure used in [7] and [19],

which is now able to identify a larger number of safe faults, thanks to a mechanism allowing to exploit the power of a commercial ATPG tool. Our method for safe fault identification is based on the following steps:

1. We identify the set of all inputs to the CPU module which will remain at a fixed value during the system operation (e.g., the Normal/Test signal). Let us call PI_{fixed} this set.
2. We perform several simulation experiments on the CPU module running the actual application and with different but realistic data input sequences and using toggle activity to identify the set $FF_{possibly-fixed}$ of flip flops which never toggle.
3. We focus on $FF_{possibly-fixed}$, and manually check whether any of the flip flops in this set may possibly toggle if a different sequence of input data and events is considered. The remaining set of flip flops, called FF_{fixed} is composed of those flip flops that will never toggle in the operating conditions.
4. We resort to an ATPG tool to identify the faults in the combinational logic of the processor that become untestable once the constraints coming from the fixed values of the PI_{fixed} and FF_{fixed} signals are applied. In other words, we specify the inputs of the combinational logic whose value always remains fixed during the operational phase, and ask the ATPG tool to identify untestable faults under these conditions. These faults correspond to safe faults for the system.

The reader should note that in [7] and [19] the last step was performed resorting to a simple topological analysis of the effects of the fixed values in the combinational logic: the analysis identified for each gate the possible untestable faults caused by any fixed value on the inputs of the fault. Hence, the analysis was only able to identify redundant signals or gates in the circuit leading to untestable faults. To perform the same step, in this paper we resort to an ATPG tool, so that a larger number of safe faults can be identified, taking into account the constraints on the input signals of the combinational logic. This allowed us to increase by about 3% the number of safe faults identified by this step with respect to the results presented in [19]. The usage of a commercial tool also makes the applicability of the proposed method easier. It is important to underline that our method cannot identify all safe faults in the system. However, we claim that it can identify a significant number of them and represents a first step towards the automation of the whole safe fault identification procedure, thus contributing to significantly reducing its cost.

C. Results

We implemented a tool based on a set of TCL scripts interacting with a logic simulator to implement the procedure described in the previous sub-section. The required time to run the simulation campaign to gather the data for the Toggle analysis and to process them to extract the list of Safe faults (including the ATPG step) is in the order of a few hours. By using the same commercial ATPG tool we also identified the number of structurally untestable faults in the OR1200, which amounts to 80. Following the proposed procedure and referring to the environment and application code of the OBC-S board, we identified a set of safe faults in the OR1200 processor, as reported in the second column of Table III. We also computed the Safe Fault Coverage (*SFC*) for the Functional and Structural tests (columns 4 and 5), defined as:

$$SFC = \frac{\#detected\ faults}{\#faults - \#safe\ faults}$$

The reported results show that:

- The number of safe faults is relevant, accounting for about 13% of the whole stuck-at fault list.
- The percentage of safe faults varies widely from one module to another. It is about 20% for modules such as *mult_mac* and *sprs* (dealing with special purpose registers, which are not significantly used by the application). It is also significant for modules such as *if*, *genpc* and *rf*, which are not fully used by the application code.
- The SFC figure achieved by the structural test procedures is quite high (taking also into account the observability constraints of the test environment) and allows (combined with some further test techniques implemented at a higher level) to fully match the reliability requirements for the MaMMoTH-Up system.
- We are still working to improve the achieved SFC by developing some test procedures targeting the few small modules which are not well covered by the current test procedures, resorting to more sophisticated solutions (e.g., triggering some exceptions or moving to supervisor mode to test the special-purpose registers) which can only be used before the launch.

Table III. Safe stuck-at fault coverage (SFC) of functional and structural tests after untestable analysis on OR1200.

Module	Safe faults	Safe Faults w.r.t. Total SA faults	SFC	
			Functional Test	Structural Test
CPU	16,183	12.98 %	36.88 %	84.41 %
genpc	425	8.66 %	66.57 %	63.24 %
if	204	9.00 %	55.57 %	76.74 %
ctrl	13	0.30 %	71.74 %	80.42 %
rf	5,550	14.21 %	39.60 %	92.89 %
opmux	41	1.62 %	92.00 %	96.47 %
alu	75	0.51 %	46.28 %	78.54 %
mult_mac	7,861	19.95 %	17.38 %	97.04 %
sprs	1,070	19.38 %	10.31 %	44.41 %
lsu	7	0.26 %	67.79 %	66.16 %
wbmux	0	0.00 %	69.29 %	78.83 %
freeze	7	5.55 %	79.83 %	79.51 %
except	912	13.58 %	18.35 %	21.85 %
cfgr	18	7.76 %	0.00 %	0.00 %

V. CONCLUSIONS

This paper deals with the adoption of COTS components in the design and manufacturing of systems to be used on a launcher. We focused on the MaMMoTH-Up system to be used on board the Ariane5 launcher, which represents a testbench for developing a suitable design and manufacturing flow compatible with the adoption of COTS components. In particular, we focused on the test of the CPU core used within such a system, showing first that the functional test is not able to achieve a sufficient test quality, while structural SBST test procedures can be much more effective. We also focused on the identification of safe faults, i.e., those faults that cannot produce any failure due the hardware and software constraints provided by the application environment. We proposed a semi-automated method able to significantly reduce the cost and effort for safe faults identification, showing that the method can identify a significant number of safe faults. We reported experimental results on the

OR1200 processor core used within the MaMMoTH-Up system. Although the proposed method has been experimentally evaluated referring to stuck-at faults, only, the same approach can be adopted to deal with other fault models (e.g., transition delay faults, or bridges), if required. We are currently working towards the development of further improved techniques for safe faults identification and towards a new and more effective release of our SBST procedures.

ACKNOWLEDGEMENTS

This work has been supported by the European Commission through the Horizon 2020 Project No. 637616 (MaMMoTH-Up).

REFERENCES

- [1] S. Avramenko; M. Sonza Reorda; M. Violante; G. Fey; J. -G. Mess; R. Schmidt, "On the robustness of DCT-based compression algorithms for space applications", 2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)
- [2] Michel Pignol, "COTS-based applications in space avionics", 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)
- [3] <http://www.mammoth-up.eu/>
- [4] <https://openrisc.io/>
- [5] UTE FIDES guide 2009, Edition A, September 2010
- [6] M. Psarakis et al., "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19
- [7] R. Cantoro, A. Firrincieli, D. Piumatti, E. Sanchez, M. Sonza Reorda, M. Restifo, "About functionally untestable fault identification in microprocessor cores for safety-critical applications", IEEE Latin-American Test Symposium (LATS), 2018
- [8] N. Kranitis; A. Merentitis; G. Theodorou; A. Paschalis; D. Gizopoulos, "Hybrid-SBST Methodology for Efficient Testing of Processor Cores", IEEE Design & Test of Computers, 2008, Volume: 25, Issue: 1, pp. 64 – 75
- [9] J.-G. Mess, R. Schmidt, G. Fey, "Adaptive Compression Schemes for Housekeeping Data", 2017 IEEE Aerospace Conference
- [10] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores", Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE), Mar. 2013, pp. 1462-1467
- [11] W. M. Globe, "Control Systems Safety Evaluation and Reliability", third edition, ISA, ISBN 978-1-934394-80-9
- [12] J. Raik, H. Fujiwara, R. Ubar, A. Krivenko, "Untestable Fault Identification in Sequential Circuits Using Model-Checking", Proc. IEEE Asian Test Symposium, 2008, pp. 21-26
- [13] Syal, M.; Hsiao, M.S., "New techniques for untestable fault identification in sequential circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 5, no. 6, 2006, pp. 1117 – 1131
- [14] H.-C. Liang; C. L. Lee; Chen, J.E., "Identifying Untestable Faults in Sequential Circuits", IEEE Design & Test of Computers, Vol. 12, No. 3, 1995, pp. 14-23
- [15] W.-C. Lai; Krstic, A.; Kwang-Ting Cheng, "Functionally testable path delay faults on a microprocessor", IEEE Design & Test of Computers, vol. 17, no. 4, 2000, pp. 6-14
- [16] A. Riefert; R. Cantoro; M. Sauer; M. Sonza Reorda; B. Becker, "A Flexible Framework for the Automatic Generation of SBST Programs", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016, Volume: 24, Issue: 10, pp. 3055 – 3066
- [17] David E. Long, Mahesh A. Iyer, Miron Abramovici, "FILL and FUNI: algorithms to identify illegal states and sequentially untestable faults", ACM Transactions on Design Automation of Electronic Systems (TODAES), v. 5, n. 3, pp. 631-657, July 2000
- [18] Daniel Tille, Rolf Drechsler, "A fast untestability proof for SAT-based ATPG", 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems, pp. 38-43, April 15-17, 2009
- [19] S. Carbonara, A. Firrincieli, M. Sonza Reorda, J.-G. Mess, "On the test of a COTS-based system for space applications", 24th IEEE International Symposium on On-Line Testing and Robust System Design, 2018, poster session
- [20] Jaroslav Borecky; Martin Kohlik; Pavel Kubalik; Hana Kubatova, "Fault Models Usability Study for On-line Tested FPGA", 14th Euromicro Conference on Digital System Design, 2011, pp. 287-290