On-line Testing for Autonomous Systems driven by RISC-V Processor Design Verification

# On-line Testing for Autonomous Systems driven by RISC-V Processor Design Verification

Annachiara Ruospo, Riccardo Cantoro, Ernesto Sanchez
*DAUIN, Politecnico di Torino*
Torino, Italy
{annachiara.ruospo, riccardo.cantoro, ernesto.sanchez}@polito.it

Pasquale Davide Schiavone*,Angelo Garofalo†, Luca Benini*†
*ETH Zurich, Switzerland**
*Università di Bologna, Italy†*
{pschiavo@iis.ee, lbenini@iis.ee}.ethz.ch*
{angelo.garofalo}@unibo.it†

***Abstract*—In the last decade, a growing number of electronic devices have been designed to be deployed in safety-critical autonomous systems. Many application domains, such as autonomous vehicles, robots, nano-drones, are exploring artificial intelligence solutions to handle the increasing computation requirements. Besides, due to their safety-critical application scenarios, they are demanding for even more reliable and advanced systems. These requirements clearly entail a growing complexity in modern processors and System-on-a-Chip design, leading to new efforts in verification and testing phases. These new devices must be also compliant with emerging functional safety standards that regulate their usage during the entire lifetime. The main intent of this work is to improve the reliability of autonomous systems, providing a strategy to link the verification methodology with the testing one. Starting from an almost exhaustive verification set, it is possible to derive a different set of patterns intended for on-line testing. This achievement is gained by taking into account the constraints due to the final system application and the common requirements of the embedded devices used in autonomous systems. Experimental results are provided on an open-source RISC-V processor assembled on an autonomous nano-drone.**

## I. Introduction

Autonomous Systems (AS) have been a subject of great interest over the last decade. Several domains, ranging from automotive industry, transport robotics, smart homes, up to Internet-of-Things devices are moving to autonomous solutions with a growing computational complexity. One of the biggest area being involved in this transition regards Unmanned Aerial Vehicles (UAV) or more commonly referred to as drones. It is not always possible or wanted to have a pilot controlling the drone: with the market growing faster, there is the need for drones to be operated autonomously [1]. Recently, UAVs have been involved in several critical tasks such as crisis situations, safety inspections and search operations. However, the real challenge is that autonomous systems must operate in areas shared by humans most of the time. Within the next couple of years [2], autonomous vehicles are expected to completely share the road with human drivers.

The increasing impact with autonomous systems in our daily life underlines the need to guarantee that these systems behave correctly even during their mission life. For this purpose, two big issues need to be addressed: AS Reliability and AS Dependability. In the context of UAVs, dependability studies and analysis for autonomous systems are presented in [1] [3]. In [1], the authors show a holistic approach for developing a dependable autonomous systems based on a cage that monitors the systems and its correctness at operation time. They argue that, since autonomous systems are exposed to constantly changing and uncertain environments, the system cannot be completely designed, constructed and tested only during the development phase.

Concerning the reliability aspect, one of the most promising techniques present in literature which is used to increase the device functional safety capability is called on-line testing. On-line testing can be performed resorting to hardware-based or software-based approaches: the former consists in inserting specific hardware modules in charge to monitor the different elements of the devices (e.g., memory, processor core) during the system power-on or power-off. It can reach really high fault coverage at the cost of being invasive and costly. The latter has been first proposed in 1980 by Thatte and Abraham and improved in several research groups [4] [5]. It consists in a software library made of several test programs. To reach a given fault coverage, the processor core is asked to execute each of these periodically, during the power-on, the power-off, to test the processor core or peripherals around it. In this second solution, it is not necessary to add additional hardware, even though it is needed to allocate some flash memory space to store the software library. The main features and constraints of an on-line software test library are discussed in [6].

The end goal of this paper is to improve Autonomous Systems Reliability, through the exploitation of an on-line software test library tied to the running application software for the device in field testing. This work presents a methodology on how to derive an on-line software test library starting from a set of verification programs of a processor core. This step is performed by a tool generator that, making use of its internal templates and an Instruction Set Architecture (ISA) dependent database, yields the final test programs. Moreover, the whole methodology starts from a preliminary process of static analysis of the circuit in order to remove those faults that cannot produce a failure during the device lifetime: the *Safe Faults*. Reducing the fault list to the only faults excited by

the running software application, means decreasing the library memory footprint as well as the effort required to raise the library fault coverage.

Experimental results are gathered on an open-source RISC-V core embedded on an autonomous nano-drone [7]. The work experimentally demonstrates that it is possible to automatically create test programs that fit for a given application software with a high fault coverage. Two examples are provided for two different processor units: the Multiplier with a 92% of fault coverage, and the Hardware Loop Controller with about 80% of fault coverage.

The rest of the paper is organized as follows: Section II provides an overview on the main features and constraints of an Autonomous System along with a background on the Software Test Library and the concept of Safe Faults. Section III describes the proposed approach, detailing all the steps necessary to get the final results. Section IV discusses the case study, showing the target application running on the nano-drone and the architecture modules selected to demonstrate the validity of the method. Section V details the experimental setup and Section VI shows the experimental results. To conclude, Section VII sums up the project and provides hints for future work.

## II. Background

In this section, to ease the understanding of the methodology, the key concepts of the work are deepened.

### A. Autonomous Drones Constraints

Autonomous Systems stem from a basic concept: performing their tasks autonomously even in unknown environments while at the same time satisfying dependability and reliability requirements. Typically, most of the tasks that an AS must schedule are performed by an external server that supports the system elaboration. However, as reported in [8], it is possible to implement a fully autonomous UAV by resorting to an ultra low power system able to run an end-to-end closed loop visual pipeline that exploits deep learning algorithms. An autonomous drone has to fit within stringent constraints. The most important requirement for a drone is the ability to adaptively avoid obstacles in order to conclude the mission without endangering people. Moreover, the battery has to be large enough to ensure a sufficient long flight [1]. From an implementation point of view, the drone application software is required to respect the on-chip and onboard resources. The drone application software is typically executed on a low-power device with a limited amount of memory and a reduced power budget.

### B. Software Test Library

A Software Test Library (STL) is one of the most popular solutions adopted by the industries concerning in-field testing. If the in-field testing is performed during the mission life (i.e., when the system is active and running), it is referred to as on-line testing. It consists of a set of Software-Based Self Test (SBST) programs; each test program is scheduled as a normal system task, along with the mission application, preferable during the IDLE phases. The coexistence with the mission application introduces very strong limitations. The main guidelines that a test program, intended for on-line testing, has to follow are:

- No alteration of RAM memory.
- No alteration of Control and Status Registers.
- No interrupts or exceptions.

### C. Safe Faults

Safe Faults identify a set of faults that can never produce a failure in the system. Since they do not contribute to the Failure Rate, they should be removed from the faults list. In automotive domain, the standard ISO 26262 describes them as "safe faults application dependent".

## III. Proposed Approach

This section describes the different steps composing the proposed methodology. The overall framework is made of two main blocks (detailed in the following subsections):

- Autonomous Systems Safe Faults Classification.
- On-Line Test Set Generator.

The former aims to identify all the existing safe faults given the target application software. The latter inherits the fault lists and produces as a result the set of test programs tailored for the considered system. This generation step exploits an already existing verification test set to extract the basic functional structure of the programs. Based on these inputs and following some internal rules, it yields the final test programs.

### A. Autonomous Systems Safe Fault Classification

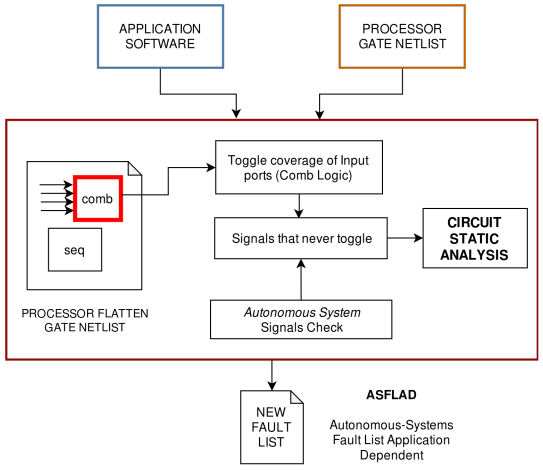


Fig. 1. Autonomous Systems Safe Fault Classification Process

The safe fault identification process aims at putting emphasis only on those faults that could influence, i.e., resulting in a failure, the behaviour and the outcome of a software application. By means of a circuit *Static Analysis*, the safe fault process is able to differentiate between faults excited by the running software application and those that are never stressed during the mission. Due to the complexity of the

analysis, recently researchers have proposed semi-automatic and automatic [9] solutions intended for general application software domain. The proposed fault classification is mainly based on [9] but, in order to be compliant with autonomous systems constraints, introduces additional steps on that basic flow. The complete flow is depicted in Figure 1. The idea relies on identifying the set of inputs of the CPU that remain at a fixed value during the system operation. Once these signals are picked out, they are propagated as constraints to the cone of combinational logic, to extract the faults relative to the bounded gates. As a result, the safe faults will be identified and removed from the CPU fault list.

To determine all the safe faults for the application software, the proposed process requires two inputs: the gate netlist of the processor core and the target application software. Therefore, the following steps are performed:

1) The hierarchical gate netlist is flattened in order to split the combinational part of the circuit from the sequential one.
2) To identify those signals that are fixed, the toggle activity of the input ports of the combinational logic is registered when the target application software is running. Then, a logic value (0 or 1) is assigned to these signals.
3) Before creating the final list of fixed signals, a tailored check is performed in order to identify all those signals belonging to special units that must not be included in that list. When dealing with Autonomous Systems, even though a signal is fixed during the execution of the application software, it does not mean that it will be fixed in any case. For instance, signals that handle interrupts coming from GPIOs, cannot be set to a fixed value, since they can change in very particular scenarios during the mission mode. Moreover, all the registers devoted to keep the processor status, even if never toggle during the application run, could change their value due to a processor unexpected exception.
4) A *Static Analysis* of the circuit is performed by fixing the constant value of the signals that never toggle. This process is useful to identify all the faults in the combinational logic of the processor core that, due to that constaints, become untestable.
5) At the end of the *Static Analysis* process, all the faults classified as undetectable and untestable will be considered safe and therefore could be removed from the processor fault list.

As output, the *Autonomous Systems Fault Classification* process provides an Autonomous Systems Fault List Application Dependent (ASFLAD), the new fault list which will be the target of the next test program generation process.

### B. On-Line Test Programs Generator

Starting from the new fault list, the following step of the proposed approach aims to create test programs suited for the target software application. After the *Autonomous Systems Safe Fault Classification* process, the amount of faults belonging to each processor unit (e.g., load and store, ALU, multiplier) changes. Considering this, the on-line test programs generator awards higher priority to those units with the higher amount of faults, i.e., units that are used more frequently by the application software. Then, to create test programs, it exploits an already existing verification test set to derive the basic structure of a program used to verify the functionality of a processor module or a set of rules in earlier phases. Clearly, the structure of a test program is quite different from a verification one. For this reason, the generator, named *ver2test*, takes only the skeleton structure from verification programs and internally applies some changes, resorting to internal templates as well as to an internal ISA-dependent database. The goal of the templates is to add common test program structures to the verification program skeleton. Indeed, to be effective, a testing program, must comply with the following structure:

- *Stack frame creation*: The content of the registers must be saved before starting the execution.
- *Register initialization*: Registers are initialized with patterns random or derived from an ATPG process.
- *Core of the program*: In the proposed approach, this part is derived from verification programs.
- *Store of the signature*: In order to examine programs behaviour in case of faults, register values are accumulated on a signature and stored in memory. In [10], the authors provide examples of signature creation for target modules.
- *Stack frame destruction*: The content of the registers is restored and the stack frame destroyed.

The reason behind the creation and destruction of the stack frame is linked to the signature usage: when executing a test program, the processor state has to be preserved to avoid signature corruption. Contrarily to test programs structure, a verification test program is lacking of signature mechanisms. The idea behind the adoption of an already existing test set comes from the advantages that every verification program holds. From a functional point of view, it is targeted for perfectly excite the unit under consideration specially in its corner cases. From a testing perspective, these corner cases could be really helpful to reach hard-to-cover circuit areas. Moreover, the other advantage with verification programs is that they commonly feature really high code coverage, meaning that almost all the instructions of the ISA are covered.

For the sake of completeness, what *ver2test* is asked to do is the following: reading the fault list obtained from *Autonomous Systems Safe Fault Classification* process and selecting those modules with the highest number of faults. Then, based on an ISA-dependent database, it selects specific programs from the verification test set and applies its templates to create an on-line test program. The ISA-dependent database contains a list of files; each one is related to a processor unit (e.g., ALU, multiplier, divider) and includes all the instructions belonging to the Instruction Set Architecture which are able to excite that module. The generator behaves differently according to the topology of the module. In case of *Arithmetic* units (e.g., adder, shifter, multiplier) *ver2test* select from verification

programs each occurrence of each instructions (following its database) and creates the final testing programs, by employing its internal templates. When dealing with *Control Modules* devoted to managing the pipeline, forwarding paths, or special units dedicated to Exceptions or Load and Store, the proposed approach requires a small effort from the verification team during the creation of their functional programs. The idea is to aid *ver2test* quickly finding the core of the program by simply adding a special label at the beginning and at the end of the skeleton program.

The reader should note that the approach can be applied to any ISAs, any processors and any kind of application software running on a system.

## IV. Case Study

To experimentally demonstrate the effectiveness of the proposed methodology, the following choices have been done.

### A. Application Software

The application software taken into account is a Deep Neural Network based Visual Navigation Engine for autonomous nano-drones [8]. This application has been developed under the *Parallel Ultra Low Power Platform* (PULP) project and it originates from *DroNet*, a lightweight residual CNN architecture proposed in [11] for standard-sized unmanned aerial vehicles (UAVs). In the target application software, *DroNet* has been adapted to use only onboard resources as well as to fit the computational requirements of nano-sized UAVs, such as fixed-point computation. The efficacy of the approach has been demonstrated on a parallel low-power GAP8 platform, an embedded RISC-V multi-core processor integrated on a Printed Circuit Board named *PULP-Shield*.

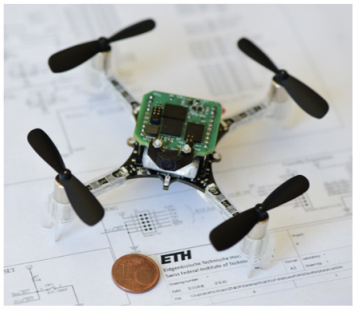


Fig. 2. PULP Shield [8]

As case study, it is shown a methodology to create an online library suited for the upgraded versions of the RISC-V cores of GAP8 computing platform. *RI5CY* [7] is one of the eight cores of GAP8. The simplified block diagram of this RISC-V core architecture is shown in Figure 3. It is an open-source, 32 bit, in-order core, based on the RV32IMFC extensions of the RISC-V ISA and the RTL is described in SystemVerilog. *RI5CY* features 4 pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX) and Write Back (WB). It has been extended with bit manipulation, HW-Loop, fixed-point, and packed Single-Instruction Multiple-Data (pSIMD) instructions to improve performance over IoT applications. In this work, *RI5CY* core has been synthesized with a 45nm NangateOpenCell Library.

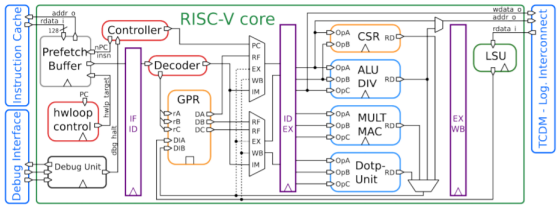


Fig. 3. RI5CY Core Architecture [7]

### B. Verification Set

In this approach, the adopted Verification Set has been developed in [12] and reaches a 90.28% of code coverage. This percentage has been achieved by exploiting an evolutionary optimizer for the test programs generation and is computed on six high level code coverage metrics: statement, branch, expression, condition, FSM state, FSM transition.

## V. Experimental Setup

This section deepens the proposed methodology by providing results of the experimental setup. All the results are gathered from *RI5CY* core described in Section IV.

The whole framework is composed of a set of tcl, python and bash scripts and exploits a commercial fault simulator for the fault simulations. The circuit *Static Analysis* has been performed relying on an commercial ATPG tool. In the performed experiments, the toggle coverage metric has been extracted resorting to Modelsim® HDL Simulation.

From this moment on, all the presented results are in line with the flow described in Figure 1. *RI5CY* core has been primarily unflattened to split sequential from combinatorial cells. From a theoretical point of view, focusing on the inputs ports of the combinational logic is equivalent to analyse the core in a given state. Therefore, to filter those signals that are fixed during the software execution, a gate-level simulation of the DNN-based nano-drones application software has been perfomed on the flatten gate netlist. The aim was two-fold: register the Value Change Dump (VCD) of the combinational logic to recover the last value assumed by the signals, and record the toggle activity of the input ports. A 56% of toggle activity on the 3,183 input ports of the combinational logic was registered. Among these ports, 1,828 toggled more than once. The remaining 1,355 were the candidates for the next *never toggle* analysis. Then, to comply with Autonomous Systems Constraints, from the never-toggle list (1,355 signals) all those related to Interrupt logic (*irq*) and Control and Status Registers (*csr*) have been removed for safety reasons. The remaining list was composed by 1,327 signals and their logic value (0 or 1) has been recovered by drawing on the VCD.

Finally, the constraints for the ATPG tool were applied. After the ATPG process, all the faults marked as *Untestable*,

*Undetectable* or *ATPG Untestable* have been considered safe and therefore removed from the *RI5CY* fault list. The reader should note that the *ATPG Untestable* class was not due to abort conditions: the ATPG process has been performed several time increasing time to time the abort time.

By following the proposed procedure and by referring to the application code of the DNN-based nano-drone, about 47% of safe faults in *RI5CY* core has been identified. Table I summarizes this result.

TABLE I
STUCK AT FAULTS BEFORE AND AFTER THE AUTONOMOUS SYSTEMS SAFE FAULTS CLASSIFICATION PROCESS (ASSFC)

| RI5CY core | Before ASSFC | After ASSFC |
|---|---|---|
| Stuck-at Faults | 268,488 | 140,206 |

As evidenced in Table II, the reduction due to the nano-drone application software varies considerably among the units: the reduction is negligible for the Load and Store but significant for Pulp Memory Protection. This means that the former is greatly used by the software while the latter is completely unexcited. The On-Line Test Program Generator (ver2test) starts from this new fault list and selects those units with the highest amount of faults (e.g., Execution Stage and Decode Stage). According to its internal ISA-dependent database and the selected modules, *ver2test* adopts its templates to produce test programs.

TABLE II
STUCK AT FAULTS DETAILED FOR RI5CY CORE UNITS BEFORE AND AFTER THE AUTONOMOUS SYSTEMS SAFE FAULTS CLASSIFICATION PROCESS (ASSFC)

| RI5CY core Faults | Before ASSFC | After ASSFC | Reduction |
|---|---|---|---|
| cs_registers_i | 29,657 | 17,330 | 42% |
| ex_stage_i | 69,214 | 50,533 | 27% |
| id_stage_i | 66,348 | 52,783 | 21% |
| if_stage_i | 18,618 | 13,345 | 29% |
| load_store_unit | 5,774 | 5,260 | 9% |
| RI5CY_pmp_unit | 77,784 | 425 | 99% |

## VI. RESULTS

To experimentally demonstrate the procedure, two units belonging to the selected modules (i.e., Execution Stage and Decode Stage) have been chosen: the Multiplier and the Hardware Loop Controller.

The multiplier unit lies in the Execution Stage. It holds 36,543 stuck-at faults but, after the ASSFC process, the 35% has been considered safe. Thus, the amount of remaining faults is equal to 24,063. To create an on-line test program suited for the system, *ver2test* makes use of its internal database looking for the multiplier file (rv32m_multiplier.txt). This file contains a list with all the instructions linked to the unit. As shown in Figure 4, each of these is sought in the verification set and then expanded in a fixed block.

Typically, to be exhaustively tested, an arithmetic unit requires more than one patterns per instruction. The generated block follows this structure: first randomly initializes its source

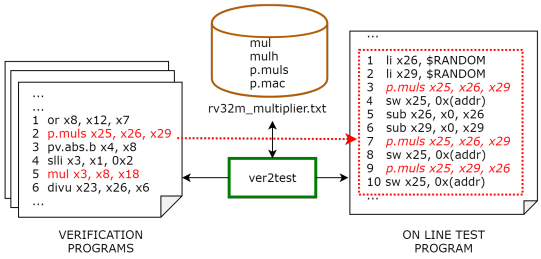


Fig. 4. Ver2test Procedure for the Multiplier Unit

registers (lines 1 and 2), then, negates their contents (lines 5 and 6) and finally flips the order (line 9). Since the Verification set features high code coverage, the final test program includes all the instructions linked to the multiplier block, and for each of them, 3 blocks are produced. Once ready, this test program has been fault simulated employing a commercial fault simulator, reaching a 92.26% of fault coverage.

TABLE III
VER2TEST GENERATED PROGRAM

| RI5CY | Instructions | Blocks/Instructions | Fault Coverage |
|---|---|---|---|
| mult_i | 66 | 3 | 92.26% |

The Hardware Loop Controller is inside the Decode Stage and owns 3,392 faults. The Hardware Loop logic aims at increasing the efficiency of small loops: the Instruction Set is extended with a set of instructions able to execute a piece of code multiple times, without resorting to branches or a counters. Following the ASSFC process, only 1,576 faults remain to be covered: the 54% has been classified as safe. As described for the multiplier, *ver2test* taps into its internal database looking for the target module. Being a non arithmetic logic module, *ver2test* needs to derive not only the single line of code but a logic block with a clearly defined beginning and end. In the drafting of verification program, it is asked to specify this cut by means of labels (e.g. TESTON-TESTOFF as in Figure 5) or defines.

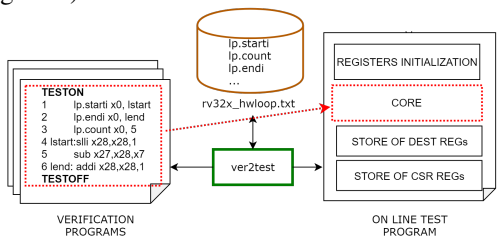


Fig. 5. Ver2test Procedure for the Hardware Loop Controller

The resulting on-line test program covers all the Hardware Loop instructions and reaches a 80.41% of fault coverage.

TABLE IV
VER2TEST GENERATED HWLOOP PROGRAM

| RI5CY | Instructions | Fault Coverage |
|---|---|---|
| hwloop_i | 8 | 80.41% |

### A. Testing Overhead: when schedule the library

When developing an on-line test set it is essential to evaluate the overhead introduced by the library test programs.

As mentioned before, they are executed as normal tasks interleaved with the running application software. This paper clearly exposes an on-going work which purpose is to deliver the entire on-line library covering all the processor modules. The IEC 61508, an international standard for managing *Functional safety of electrical/electronic/programmable electronic safety-related systems* does not expressly fix an interval time to schedule the on-line test library. However, a reasonable example comes from the autonomous road vehicles domain: the ISO 26262 standard is an adaptation of the IEC 61508 for Automotive Electric/Electronic Systems and claims that the Diagnostic Time Interval (DTI) for fault detection in a CPU can be around 10ms [13]. For the target autonomous nano-drone it is reasonable to execute the on-line test library after each frame elaboration. The application can scale up to 18 fps at 250MHz [8], thus, it can elaborate a frame about every 55 ms. Since the average execution time of a single test program is about 10us @250MHz (2,500 clock cycles), we estimate that the overall test library will last from 500us to 600us (@250MHz). By enabling the library during the normal execution of such application, the user can tune operational conditions (system frequency and voltage) to choose between same performance or same power budget according to the safety requirements of the system. For instance, supposing to execute a test phase after each frame elaboration (55 ms @250MHz), in order to keep the same frame-per-second (fps), the frequency of the system has to be increased up to 253MHz to avoid that the introduced overhead down-performs the application. This extra power overhead and shorter battery lifetime is traded by a higher level of reliability ensured by the on-line test library. Due to the coexistence between the application and the safety mechanism, autonomous systems have the ability to take appropriate actions when some functionality is lost due to failures.

## VII. Conclusions

Autonomous systems are on the rise and it becomes of a paramount importance to test these systems and ensure their safe behaviour during their mission life. Providing proper solutions for safety and reliability is currently a crucial challenge for the long-term societal acceptance of autonomous systems. Moreover, utilizing deep learning algorithms for training and inference introduces additional safety certification challenges for these complex devices [14]. The end-goal of this paper is to present a methodology to develop on-line test programs to be executed along with the running software application. This solution is mostly oriented to autonomous systems but is extensible to every domain. Guidelines are provided to identify the amount of safe faults, i.e., those faults that cannot produce any failure due to the hardware and the software constraints provided by the application environment. Moreover, experimental results prove that it is possible to reach high fault coverage (80% - 90%) by exploiting an already existing verification set of programs. The proposed approach aims at significantly reducing the cost and the effort for creating ad-hoc test programs by providing a fully automated solution. The experiments are reported on a RISC-V processor (*RI5CY*) when running a DNN-based visual navigation engine for autonomous nano-drones application software.

Future work aims at running the complete on-line test library in the field and on the fly, i.e., during the nano-drone mission mode.

## References


[1] A. Devos, E. Ebeid, and P. Manoonpong, "Development of autonomous drones for adaptive obstacle avoidance in real world environments," 08 2018, pp. 707–710.

[2] A. Aniculaesei, J. Grieser, A. Rausch, K. Rehfeldt, and T. Warnecke, "Toward a holistic software systems engineering approach for dependable autonomous systems," in *2018 IEEE/ACM 1st International Workshop on Software Engineering for AI in Autonomous Systems (SEFAIAS)*, May 2018, pp. 23–30.

[3] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell, "Fault tolerance in autonomous systems: How and how much?" 01 2005.

[4] 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.

[5] A. Jasnetski, R. Ubar, and A. Tsertov, "On automatic software-based self-test program generation based on high-level decision diagrams," in *2016 17th Latin-American Test Symposium (LATS)*, April 2016, pp. 177–177.

[6] P. Bernardi, R. Cantoro, A. Floridia, D. Piumatti, C. Pogonea, A. Ruospo, E. Sanchez, S. De Luca, and A. Sansonetti, "Non-intrusive self-test library for automotive critical applications: Constraints and solutions," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 920–923.

[7] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct 2017.

[8] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini, "A 64mw dnn-based visual navigation engine for autonomous nano-drones," *IEEE Internet of Things Journal*, vol. PP, pp. 1–1, 05 2019.

[9] R. Cantoro, S. Carbonara, A. Floridia, E. Sanchez, M. S. Reorda, and J. Mess, "An analysis of test solutions for cots-based systems in space applications," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2018, pp. 59–64.

[10] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. Reorda, S. De Luca, R. Meregalli, and A. Sansonetti, "On the in-field functional testing of decode units in pipelined risc processors," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 299–304.

[11] A. Loquercio, A. I. Maqueda, C. R. del Blanco, and D. Scaramuzza, "Dronet: Learning to fly by driving," *IEEE Robotics and Automation Letters*, vol. 3, pp. 1088–1095, 2018.

[12] P. D. Schiavone, E. Sanchez, A. Ruospo, F. Minervini, F. Zaruba, G. Haugou, and L. Benini, "An open-source verification framework for open-source cores: A risc-v case study," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Oct 2018, pp. 43–48.

[13] A. Nardi and A. Armato, "Functional safety methodologies for automotive applications," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 970–975. [Online]. Available: http://dl.acm.org/citation.cfm?id=3199700.3199834

[14] J. Athavale, R. Mariani, and M. Paulitsch, "Flight safety certification implications for complex multi-core processor based avionics systems," in *2019 IEEE International Reliability Physics Symposium (IRPS)*, March 2019, pp. 1–6.