Enhanced software development process for CubeSats to cope with space radiation faults

David Paiva
CISUC, University of Coimbra
Coimbra, Portugal
davidpaiva.uc@gmail.com

Raffael Lima
COENE, INPE
Natal, Brazil
raffael.sadite@inpe.br

Manoel Carvalho
COENE, INPE
Natal, Brazil
manoel.carvalho@inpe.br

Fátima Mattiello-Francisco
COEPE, INPE
São José dos Campos, Brazil
fatima.mattiello@inpe.br

Henrique Madeira
CISUC, University of Coimbra
Coimbra, Portugal
henrique@dei.uc.pt

***Abstract* — CubeSats are an established trend in the space industry. The CubeSat standard opens opportunities for rapid and low-cost access to space. The use of COTS components instead of space-hardened hardware greatly reduces the cost of CubeSat-based missions and provides the additional benefit of increasing software functionalities at a low power consumption. However, COTS components are not designed for the space environment, making CubeSats sensitive to space radiation. This means that CubeSats need additional software mechanisms to guarantee resilient behavior in the presence of space radiation. Our proposal is that such software implemented fault tolerance mechanisms must be tailored to the specific code running in each CubeSat and the logical way to achieve that is to extend the software development process for CubeSats to include the systematic resilience evaluation of software as part of the CubeSats software lifecycle process.**

This paper proposes a set of structured steps to enhance the classic software development process used in CubeSats, focusing particularly on the Verification and Validation (V&V) phase. The approach uses fault injection as an integral part of the development environment for CubeSats software and includes three major steps: a) sensitivity evaluation (verification) of software in the presence of faults caused by space radiation, b) strengthen of the software with targeted software implemented fault tolerance (SWIFT) mechanisms and c) validation of the effectiveness of the SWIFT mechanisms to confirm that the software is immune to space radiation faults. These added steps to the V&V process must be carried out during software development, as well as every time the CubeSat software has an update, or even a minor change, to ensure that the impact of faults caused by space radiation is tolerated by the CubeSat software. The paper demonstrates the proposed approach using three different embedded software running in the EDC (Environment Data Collection) CubeSat board, which is part (payload) of a constellation of satellites being developed by the Brazilian National Institute for Space Research (INPE). EDC use case provides a realistic insight on the effectiveness of the proposed steps. Our results show that the proposed approach can reduce the percentage of silent data corruption (the most problematic failure mode) from the range of 15% to less than 1% and even to 0% in some embedded software, meaning that the CubeSat software becomes immune to space radiation.

Keywords* — *CubeSats, COTS, software development, verification and validation, soft errors, fault injection, software fault tolerance techniques

I. Introduction

Nowadays, the interest in the development and deployment of CubeSats solutions has become a trend in the space industry. CubeSats are small-satellites built with up to 12 units in the shape of a cube of 10cm edge and weight of 10kg maximum, according to the CubeSat Design Specification (CDS) - a standard de facto for mechanical design and interfacing for satellites [1]. In fact, the CubeSat standard strongly reduces cost and development time of space projects, increases accessibility to space, and sustains frequent satellite launches. CubeSat-based projects place emphasis on the use of Commercial-Off-The-Shelf (COTS) components and systems. When compared with space-hardened components - specially designed to withstand the harsh space conditions - COTS present several benefits like low cost, high performance, and low energy consumption, which open opportunities to rapid develop of new space technologies and to carry out affordable space missions.

Despite these advantages, COTS components are not designed for space applications, which means they are susceptible to transient errors as a result of single event upsets (SEU) caused by space radiation. In fact, errors caused by SEU are established as the major cause of COTS components failures in space [2]. The impact of space radiation could damage COTS on a permanent basis, but the most common effect is to cause transient faults [2] that may lead the software to crash or to produce erroneous results.

Although CubeSats generally use ordinary COTS hardware (i.e., sensitive to space radiation), typical architectures of CubeSats boards [3] include several mechanisms to cope with faults caused by space radiation. Memory is typically protected through error detection and correction codes, and communication structures also use error detection and correction provided by the communication protocols and associated hardware of the communication links. Memory, in particular, represents a large silicon surface exposed to radiation, which means that protecting memory from transient bit flip errors due to space radiation is mandatory.

Fortunately, the protection of memory and communication channels against transient faults caused by space radiation is relatively easy to achieve at low cost because of the regular nature of such structures. For example, the use of extended Hamming codes [4] to assure single error correction and double error detection in the memory just requires two extra parity bits and is a frequent solution in CubeSat boards. Similarly, the use of communication protocols and techniques such as forward error correction codes [5] are effective in dealing with errors caused by transient faults in communication channels.

The big challenge is to protect the processor(s) of CubeSat boards from the effects of space radiation. Obviously, the use of space-grade processors that resist space radiation is not an option for CubeSats, as the cost of such processors is several orders of magnitude higher than the cost of common COTS processors. But, unfortunately, COTS processors are not immune to space radiation and, at the same time, the complex internal structure of processors does not allow the use of affordable data error detection and correction methods that protect uniform and regular structures such as memories and communication channels. In other words, existing CubeSat boards can deal with transient faults caused by space radiation that affect memory and communication, but the processor represents the major weakness for the reliability of CubeSats.

The obvious solution would be to rely on classic fault-tolerant architectures at the board level [6] to tolerate faults of the COTS processors in CubeSats. But these techniques represent a substantial increase of hardware redundancy, with high negative impact on the board weight and power consumption. For example, the use of duplicated processors in CubeSat boards would require a large amount of additional hardware to deal with the comparison of the two processors, no matter the concrete flavor of fault-tolerant architecture used in the board design. For example, techniques such as lock-step dual processor architectures would require the low-level comparison of the hardware signals of both processors (and, most likely, can only be used if the processors are implemented in FPGAs to have access to the internal processor structure to allow synchronization of signals). Other architectures such as symmetric multiprocessors (i.e., two or more identical processors sharing a single main memory) would also need additional hardware and have negative impact at other levels (e.g., would require a multiprocessor-aware operating system) [6].

Recent research work (PhD thesis of C. Fuchs, December 2019 [7]) proposes a novel on-board-computer architecture for very small satellites (<100kg) capable of achieving high reliability without using radiation hardened semiconductors, through the combined use of hardware and software-implemented fault tolerance techniques [7]. However, in spite of this promising research result from C. Fuchs, to the best of our knowledge, there are no fault-tolerant boards available for CubeSats, especially boards that can cope with transient faults that affect the processor, which are the major threat for the reliability of CubeSats.

The current situation in the space industry is that, in spite of the growing interest in CubeSats, this category of satellites is still considered as not adequate for high-priority and critical missions, and the reason is the low reliability of CubeSats [8]. Data from 178 launched CubeSats show that the 2-year reliability estimation ranges from 65% to 48% [8]. The detailed analysis of the results presented in [8], concerning the subsystem identified as root cause of the failure, shows that the payload subsystem contributes with modest figures (from 3% to 4%), which make sense in an analysis focused on failures of CubeSat missions with a strong incidence of DOA (dead-on-arrival), where the satellite never achieved a detectable functional state. However, we may speculate that the failure rate in CubeSat payload software could be much higher, especially considering transient failures in the payload software that, apparently, has not been considered in [8].

In this paper we propose a pure software implemented solution that allows us to improve the reliability of existing CubeSats without requiring any change or extra hardware in the CubeSats boards currently available. Our approach takes into account the fact that the impact of faults caused by space radiation at processor level is highly dependent on the actual software running in the CubeSat, as the error propagation phenomena and the translation of the erroneous behavior caused by faults into critical failure modes depend on low-level features of the code such as the data structures and code constructs. Abundant fault injection literature shows that depending on the actual code, the effect of faults could be relatively minor or could be devastating. This fact can be attested in many fault injection papers as reported (and condensed) in periodic surveys and fault injection papers (see [9], [10], [11]).

In particular, a project involving NASA JPL [12] reported results from an injection campaign on a NASA COTS-based payload system for onboard processing of scientific data and shows that the percentages of the different failure modes are quite dependent on the software running in the system at the moment when the faults were injected. This difference reaches up to 45% in some failure modes, particularly in potentially dangerous failure modes such as “silent data corruption”, in which the radiation induced faults cause erroneous software results but do not activate any error detection mechanism available in the system. Similar results have been obtained in a recent and quite comprehensive fault injection study [13].

This dependency of the impact of transient faults on the actual software running in the CubeSats suggests that software implemented solutions must be instantiated at all levels of the software development lifecycle, considering both system software (operating systems, libraries, etc.) and application code. This is precisely the goal of the present paper that proposes an additional set of steps for the development of software applications for CubeSats. The contributions of the paper are as follows:

- Proposes an extension of the software development process for CubeSats using fault injection as an integral part of the development setup to **verify** the sensitivity of the software to space radiation induced faults and **validate** subsequent iterations of the software enhanced with software implemented fault tolerance mechanisms.
- Presents the proposed extension as a set of additional steps to the V&V phase. These steps include 1) sensitivity analysis, 2) software enhancement with fault tolerance techniques and 3) validation of the final software resilience. Although this extension is presented and discussed in the paper as part of the traditional waterfall V model, the proposal is largely agnostic concerning the software development process and can be used in agile methodologies as well.
- Applies the proposed methodology to a concrete CubeSat board and shows that the effects of transient faults induced by space radiation can be reduced to nearly zero using the proposed approach.

The structure of the paper is as follows: next section presents a brief state of the art on software development practices for CubeSats; section III presents the proposed approach; section IV describes the use case using a real CubeSat board, including the application do the proposed approach and discusses the results; and section V concludes the paper and outlines future work.

II. Software development practices for CubeSats

The advent of the CubeSats has made the development cycle of small satellite projects much faster and cheaper than traditional space missions. The satellite structure, cabling and interfaces have been significantly simplified with the CubeSat standardization [1], but the complexity of software embedded in the satellite subsystems just increased. Thanks to technological evolution of embedded electronics, memories and satellite processors, the potential for adding functionalities implemented by software has grown [14]. Subsystems onboard CubeSat based satellites have their own software architecture and the challenge lies in the integration of the so-called software-intensive systems (SiS) in a short development time imposed by CubeSat missions [15], [16].

The focus on the concept of interoperability of SiSs aboard spacecraft is not a concern limited to CubeSats. In the last 20 years, the satellite industry evolved from viewing software (mostly developed in house) as an important aspect of the entire spacecraft, to the current trend in which satellite software results from the integration of SiSs provided by different suppliers. Efforts in the V&V process are fundamental to support the integration phase with effective tools and methods, which made the process onerous in time and resources. This is acceptable in the development cycle of traditional satellites but incompatible to CubeSat philosophy, whose project shall be much faster and cheaper [17].

Moreover, CubeSats also have caused a shift from the classic waterfall model usually adopted for space software development into incremental approaches and agile methods. Currently, the development of most CubeSat software follows such approaches. The On Board Data Handling (OBDH) typically has grown in complexity because new services are required by payloads late in the development time. More pieces of software are developed to interface, control and operate different subsystems. These new features increase the overall complexity of the satellite and increase the number of possible software defects [15], [16], making the behaviour of the software under faulty conditions caused by space radiation quite unpredictable. Considering that the success of CubeSat missions relies on the OBDH, it is essential to mitigate such weaknesses without increasing the costs of the V&V.

Regarding software assurance practices, simulation and testing are the most common activities to verify and validate CubeSats software, according to a survey conducted at NASA Ames Research Center [18]. However, even those activities do not receive due attention on CubeSats projects, as an intensive program of verification and validation cannot be accommodated into the limited budget of such projects. Despite this, according to the same survey [18], an emerging trend relies on the use of model-based design methods due to their capability to automate the creation of detailed software design from high-level graphical inputs, and then use automatic code generation to create the code. Unfortunately, the automatic code generation for complex projects is still limited.

Although fault-tolerant software methods are used for run-time monitoring in CubeSats, the use of rigid V&V methods is not a trend in current CubeSats software development due to time and budget constraints of such projects. This includes the crucial verification of possible effects of space radiation induced faults.

III. ENHANCED VERIFICATION AND VALIDATION FOR CUBESATS SOFTWARE DEVELOPMENT

A. *Context and preliminary experiments to validate assumptions*

The proposed approach uses fault injection as an integral part of the development environment for CubeSats software and includes three high-level steps:

a. Sensitivity evaluation (verification) of the software in the presence of faults caused by space radiation;

b. Enhancement of the software with targeted software-implemented fault tolerance (SWIFT) mechanisms; and

c. Validation of the effectiveness of the SWIFT mechanisms to confirm that the software is immune to space radiation faults.

These steps must be carried out during CubeSat software development, as well as every time the CubeSat software is updated, to ensure that the impact of faults caused by space radiation is tolerated by the CubeSat software. Fault injection is an establish technique and SWIFT techniques are well-known. The innovation of the proposed approach is not in the use of fault injection to evaluate the CubeSat boards (that is not effective as the impact depends much more on the software running on the board than on the CubeSat board itself), nor in the use of SWIFT to propose a fault tolerant architecture for CubeSats. The new aspect of our approach resides in the fact that fault injection and SWIFT are now proposed as integral part of the software development process for CubeSats. Fault injection is essential to evaluate the impact of SEU on the CubeSat software and SWIFT makes the software resilient to space radiation.

CubeSat boards high-level architecture can be divided in three layers, as shown in Fig. 1. The **hardware layer** is the physical part of the CubeSat composed of several COTS-based elements, such as the onboard computer, payload boards, solar panels, RF antennas, among others. The middle layer is the **system software** that includes the operating systems running in the different boards (e.g., FreeRTOS, eCos, among others) and, depending on the specific board, may include other software elements such as drivers and software designed to deal with specific sensors or actuators or to perform specific functions (e.g., to control satellite attitude). The third layer is the CubeSat **application software** that performs specific mission tasks and is developed to run on the different satellite boards, on top of the system software.

At the hardware layer, CubeSats boards use regular COTS components but the boards include several mechanisms to make them more resilient to the space environment. CubeSat boards are ruggedized with layers of resin coating for mechanical and thermal protection. Additionally, memory is protected with error detection and correction bits, as well as communication channels are also protected with error detection and correction mechanisms provided by the communication protocols and associated hardware of the communication links. Memory, in particular, is protected against transient bit-flip errors due to space radiation, as the memory chips represent a large silicon area exposed to radiation, making SEU in memory very frequent.

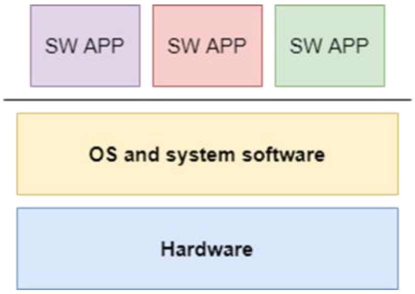

Fig. 1. Simplified view of a CubeSat board organization

As already mentioned, the protection of uniform hardware structures such as the memory and the communication channels is very simple and is in fact a common practice in standard COTS hardware boards for all sort of applications. CubeSats simply take advantage of available standard solutions such as extended Hamming codes [4] for single error correction and double error detection in the memory and forward error correction codes [5] for transient faults in communication channels. These mechanisms are well aligned with the CubeSat "philosophy" of low cost, low energy consumption, and low weight.

Protecting the processor from the SEU effects is the central problem because the processor is not a regular and simple structure. The use of space-grade processors that resist to space radiation is not an option for CubeSats because of the very high cost of such processors. The obvious solution would be to use classic fault-tolerant architectures with massive levels of redundancy, as the ones used in large-scale satellites [19] or in the aircraft industry [20], [21]. Unfortunately, these well-proven solutions are not an option for CubeSats, even if designed around COTS components, as they are expensive, heavy, and require high power consumption. Classic architectures used in avionics and in large satellites would require pairs of duplicated processors and the inherent hardware logic to vote the results from the different signals, which would ruin the simplicity and low cost of CubeSats.

The reality is that there are no fault-tolerant CubeSat boards available from manufacturers that solve the problem of transient faults in the processor at the hardware board layer and CubeSats are still regarded as very low-cost small satellites for non-critical low earth orbit (LEO) missions.

One important advantage of classic fault-tolerant techniques applied at a low architectural level (e.g., triple modular redundancy [21] or even hybrid proposals such as the recent architecture proposed in [7]) is that these techniques provide a reasonable transparent solution for the development of software applications on top of a fault-tolerant architecture. That is, the developer of application software does not need to worry about possible transient faults, as they are tolerated at the lower levels of the hardware layer or by the system software [23], [24].

Since there are no fault-tolerant CubeSats boards currently available (and they are not likely to appear in the near future because of the high cost, energy consumption, and weight imposed by hardware fault tolerance), it means that possible solutions for the transient processor faults due to SEU are not transparent for the developer of software applications for CubeSats. This is obviously a clear assumption for any proposal that attempts to solve the problem of transient processor faults in CubeSats boards through the use of SWIFT techniques, which also includes the approach proposed in this paper. The developer of CubeSat applications must be aware that the application may be affected by transient processor faults and deal with the SWIFT techniques needed to tolerate such faults (i.e., the SWIFT techniques are an integral part of the software under development). Naturally, the development of CubeSat applications will become more complex, as the application software needs to deal with both the functional aspects and the SWIFT techniques, but this is the price to pay to assure the required reliability for CubeSat applications running on simple and low-cost non-fault-tolerant boards.

The development of a library of software components that implement the skeleton of software fault-tolerant techniques is out of the scope of the present paper. However, in the context of a future industrial application of the proposed steps, it will be crucial to have a library of SWIFT methods to be used/adapted to each particular situation, in order to simplify and accelerate the development of CubeSat software capable of tolerating the hardware transient faults caused by space radiation. Of course, those techniques should be tailored to the specific software under development, as mentioned before, but a general skeleton or code (e.g., a voter that compares two inputs and signs if they differ) that can be reused could be made available in the form of reusable components available for the software development teams. This will reduce the time necessary to integrate SWIFT techniques into the code under development, making it easier and cheaper to apply the approach proposed in this paper.

The application of SWIFT techniques at the software application level to tolerate hardware transient faults, as proposed in the present paper, relies on two assumptions:

a. The system software, and specifically the operating system of the CubeSat board, is operating properly after the transient fault, allowing the correct processing of SWIFT techniques at the application level; and
b. Possible malfunctions (errors) caused by the fault can be detected by the error detection mechanisms available in the CubeSat board, so the board can be restarted to re-establish a correct state to run the SWIFT techniques and tolerate the fault.

This means that in the worst-case scenario (bullet b)) when an error is detected or the system crashes as a consequence of the transient fault, the base layers of the CubeSat (i.e, hardware and operating system) should be able to recover the system to a state from which it can operate properly. In satellite systems (and in general in cyber-physical systems) this is done through the use of classical forward recovery techniques [25], [26], [27] that bring the system to a correct state, normally resetting key elements such as the operating system. To assure this, a key feature of base layers of CubeSats (hardware and system software) is the effectiveness of the error detection mechanisms available in such layers

As mentioned, all CubeSats boards have error detection of two bits errors and correction of one bit in memory using extended Hamming code [4]. The correction of one-bit error is fully transparent, as it is processed at the hardware level, and in case of detection of errors in two bits (no correction), the error must be handled by the system software (in general, the action is to reset the system as these errors are mostly caused by transient faults due to SEU, and they disappear after reset).

Another very relevant error detection mechanism that also exists in all CubeSat boards is the watchdog timer (WDT) [28] that detects deviations of the correct software behavior that changes its timing features (most frequently, WDT are used to detect crashes). WDT can be controlled (i.e., refreshed periodically) by the system software, which makes the error detection transparent to the application software, or can be periodically refreshed by the application software. Other types of simple error detection mechanisms are associated with the memory management units of the CubeSat board and allow the detection of erroneous memory access behavior (e.g., instruction fetch outside the code segments, read/write in memory areas not available, etc.). More sophisticated (and also more effective) error detection mechanisms such as signature monitoring [29], [30], are in general not available in CubeSat boards.

Given the relevance of the assumptions mentioned above (bullets a) and b)) for the approach proposed in this paper, we decided to perform a preliminary experiment to evaluate these assumptions in faulty scenarios. The goal is to evaluate the probability of the CubeSat board (hardware layer and system software) to behave correctly after a fault, in such a way that SWIFT techniques can be applied to tolerate the faults. It is clear that SWIFT techniques can only be applied if the operating system is working properly.

This preliminary experiment consists of the injection of 10000 faults into the EDC board (a real CubeSat board) used as case study. For this experiment, the EDC board (target system) was not running any real software application. Instead, the EDC was just running the real-time operating system (FreeRTOS) and, a "dummy" task that blinked a LED light and refresh the watchdog timer counter. The idea was to evaluate the impact of faults in the system software (mainly the FreeRTOS), to evaluate whether the operating system is running properly after the fault or not.

The faults were injected following the traditional approach that emulates transient hardware faults in the processor through single bit flips in the processor registers, as proposed and used by reference fault injection works/tools in the last decades [31], [32], [33], [39]. More specifically, we used CubeSatFI [39] (fault injector dedicated to being use on CubeSats) where the faults were injected at random in all the processor registers and at a random time during the execution of the software, to emulate the random effects of space radiation.

The results obtained are presented in Fig. 2. The confidence intervals (shown in the numeric values in each bar of the chart) are calculated for 95% of confidence, using confidence intervals for proportions in binomial distributions (Bernoulli trials).

The classification of the failure modes was made based on the results obtained and includes the following failure mode types:

- **No Effect/OS OK**: The fault had no visible impact on the system. The operating system continues to work normally as expected.
- **Error detection (WDT)**: The fault crashes the operating system, but the watchdog detects this erroneous situation and restarts the system. After restarting, the system is working normally again.
- **OS CRASH**: The system crashes after being affected and the watchdog timer cannot detect it.

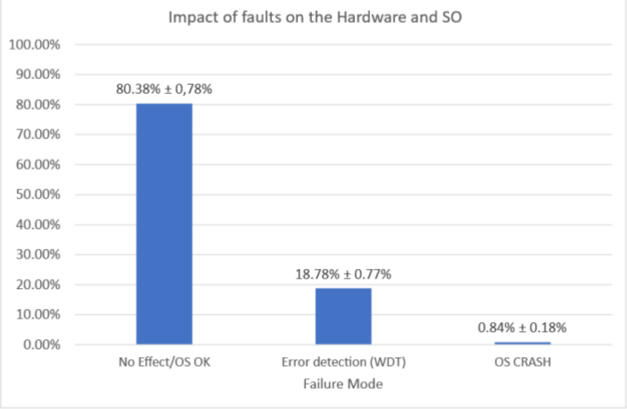

Fig. 2. Impact of faults on the Hardware and operating system

The results show that most of the faults did not affect (80.38%) the operating system, which means that the operating system continues operating properly, as expected. This result is in line with previous fault injection results [9], [10], [11], [12], [13] in other systems, as the inherent redundancy and unused resources lead to a large percentage of benign faults. It was observed that 18.78% of the faults activate the watchdog timer, assuring that after a crash the operating system can restart and back operating properly again. These two values together (**99.16%**) show that the hardware layer and error detection at the system software layers meet the assumptions described above (both a) and b)) in more that 99% of the faults, and SWIFT techniques can be effectively applied at the software application layer. This means that software developers can develop applications on top of COTS boards and use SWIFT techniques to tolerate processor transient faults due to SEU, as the probability of the operating system and the system software being operating properly after the transient fault (to allow the correct processing of SWIFT techniques at the application level) is very high (> 99% in our experiments).

It is worth noting that in this experiment the error detection available in the target system (EDC board) was only the WDT. Even so, the percentage of cases observed in which the proposed approach could not work is reduced to **0.84%**. Obviously, the inclusion of additional error detection mechanisms in the CubeSat boards could reduce even further this percentage. Even knowing that the percentages of cases where the assumptions a) and b) (see above) are met may dependent on the actual operating system and system software, we consider this result quite encouraging.

B. *Enhanced Verification and Validation Steps*

Our proposal focuses on enhancing the verification and validation of CubeSats software through a set of additional steps. These steps are intended to be the least intrusive possible on the software development life cycle used by the companies, space agencies, and other institutions that are developing CubeSat software. Since budget and time are constraints that must be considered, expensive software verification and validation activities are impossible to accommodate on such projects.

The proposed steps require a fault injection tool as part of the toolset used in the CubeSats software development process. The use of fault injection tools is quite common in the software industry [34], [10] and most fault injection tools are considered simple and affordable tools, fully in line with typical CubeSat budget constraints. In particular, fault injection tools using JTAG and the Test Access Port (TAP) such as many existing tools (e.g., [32], [33], [35]) can be easily adapted to CubeSats, as nearly all CubeSat boards are equipped with JTAG and TPA.

Fig. 3 illustrates the proposed additional steps. More specifically, our proposal does not change the previous phases of the existing software development process, but simply adds additional V&V steps after the integration test step, which is always part of the process, no matter the flavor of the software development process used by the CubeSat developer.

Step 1 - Evaluate the software sensitivity to space radiation: After integration testing the software is subject to a comprehensive fault injection campaign to evaluate the impact of SEU on the CubeSat behavior. Faults are injected in the processor registers of the target board using a random distribution (both in space - registers- and time), since space radiation tends to affect the processor randomly. This will allow us to understand the behavior of the target software in the presence of space radiation that affects the processor of the board where the software is running.

Step 2 - Strengthen the software with tailored software implemented fault tolerance (SWIFT) techniques: The results obtained in the previous step must be analyzed and the impact of the faults on the target software should be categorized into failure modes. According to the failure modes obtained, the **project manager** should decide if it is necessary enhance the software with additional SWIFT techniques to avoid failure modes such as **silent data**

corruption (erroneous output results with no error detection) or to recover the software after **crash failure modes**. This decision should be taken considering the criticality of the CubeSat mission, the resources available in the target CubeSat board, and the budget available to implement these techniques. Many SWIFT techniques can be used, from simple re-execution and voting to self-checking software [25], [26], [27], [35], [36]. If the target system has enough resources, it is extremely recommended to add SWIFT techniques to increase fault coverage as much as possible. Obviously, we are aware that including additional SWIFT techniques in the CubeSat software after a first version of the software has been through integration testing could be problematic. For fault masking techniques such as software re-execution and voting [36], [37] this task of adding this technique to existing software is relatively easy. But for other SWIFT techniques such as algorithm-based fault tolerance [37] the existing software must be largely refactored to incorporate the SWIFT technique.

Step 3 - Validate the effectiveness of the SWIFT techniques: After the software is strengthened with additional SWIFT techniques, it must be submitted to regression testing (using a test suite developed in earlier stages of the software development lifecycle) to assure that the functional requirements (and also non-functional requirements such as response time) are still met. The validation of the effectiveness of SWIFT is then performed through a fault injection campaign similar to the one run in step 1. That is, the process enters the cycle proposed in Fig. 3 until the desired software resilience in the presence of transient faults is achieved. The objective is to have a flexible way to enhance the resilience of the CubeSat software to cope with transient faults to reach the software resilience level that is necessary for the CubeSat mission and validating the results through fault injection in the actual software under development. This incremental approach (i.e., just enough software fault tolerance) seems more appropriate to CubeSat constraints, avoiding the drawback of generic (and massive) software fault tolerance architecture that costs resources and increase the impact surface of space radiation on the software behaviour as observed in [38].

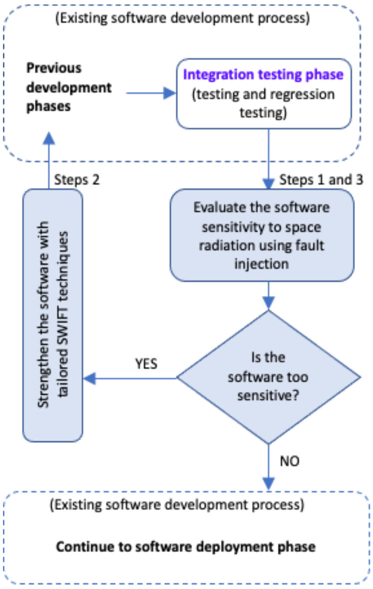
(Existing software development process)
Previous development phases
Integration testing phase (testing and regression testing)
Steps 2
Steps 1 and 3
Evaluate the software sensitivity to space radiation using fault injection
Strengthen the software with tailored SWIFT techniques
YES
Is the software too sensitive?
NO
(Existing software development process)
Continue to software deployment phase

Fig. 3. Additional steps for the CubeSat software development process

The proposed steps should be included in the software development process used in the CubeSat development project. If the project follows the classic V-Model, the fault injection evaluation of the software sensitivity to SEU should be included after the integration testing (right-side of the V). Obviously, if the CubeSat project follows an agile process (this is a growing trend in CubeSats development), the proposed steps should be performed each time that the software has a considerable increment. Since the impact of SEU-induced faults depends on the actual software that is running on the CubeSat, every time the software changes, it is crucial to perform the proposed additional V&V steps. In fact, these steps are quite in line with test-driven development (TDD) used in agile development processes, where the software requirements are converted into test cases and each software increment aims to pass the new set of test cases. After the test pass, the code is refactored, and the test suite is run again to assure that no existing functionality is broken. This cycle is repeated for each new functionality. Similarly to TDD, when the CubeSat software has a major or even a minor change, the proposed V&V steps should be executed to evaluate the resilience against SEU-induced faults, and the software is considered fully developed when it meets the safety and dependable requirements to tolerate space radiation.

An important aspect for the actual application of the proposed approach is the availability of fault injector tools such as CubeSatFI [39] as part of the software development environment to allow the execution of fault injection campaigns in an easy and automatic way (and at low cost).

IV. Use Case: EDC CubeSat Board

This section presents a use case of the proposed enhanced V&V approach using the Environmental Data Collector (EDC) [40], a CubeSat payload board for the Brazilian Environmental Data Collection System (SBCDA). Is not worth mentioning that this payload is going to be used in all the future CubeSats from the CONASAT-project [41].

A. *The EDC CubeSat board*

The CubeSat platform is a 1U (i.e., the satellite has a cubic shape with edges of 10 centimeters) and has a classical CubeSat hardware architecture comprising the onboard computer (OBC), payload boards (EDC in this case), two UHF antennas, a UHF transceiver, an electrical power system (EPS), and a battery pack. Fig. 4 shows the main blocks of the satellite hardware architecture.

The EDC board is a new payload developed to meet the demand for a signal receiver CubeSat-compatible to provide onboard signal processing. The EDC design uses only COTS components, which makes the EDC less reliable than a classic space grade transponder. The board was designed by a division inside INPE and was produced by an external company under direct quality control of INPE, while the software for the EDC board has been developed in house by the EDC software team. This is a typical arrangement for the development of CubeSats for real (and serious) missions. Furthermore, the EDC software team works with the team responsible for the development of the flight software for the onboard computer (OBC) to carry out the integration of the EDC payload in the satellite.

The UHF antenna - Payload is dedicated to receiving signals from the data collection platforms (DCP) and is connected to the EDC, while the UHF antenna - TMTC is used for communication with the receiving stations (RS) and is connected to the UHF transceiver. The UHF transceiver is the subsystem responsible for receiving and transmitting the telecommand (TC) and telemetry (TM), respectively. The EPS subsystem supplies power to the entire platform through six solar panels and several voltage converters. The platform also has a battery pack with a capacity of 10.2 Wh. The OBC is responsible for configuring, controlling, and commanding the operation of all satellite subsystems. The telecommands received from an RS are decoded in the OBC to control the subsystems onboard the satellite. The OBC is also responsible for monitoring the overall health of the satellite. Health assessment can be performed in several ways, depending on the subsystem being assessed. Telemetry sensors are used to verify that the parameters of a given subsystem are acceptable (such as temperature or voltage level). Telemetry data collected from each subsystem is also stored for transmission to an RS. The OBC acts as the I2C bus master for transmitting commands to the EPS subsystems, UHF antennas, and UHF transceiver. The flight software implements in the OBC a routine of commands and requests to control the data processed by the EDC. This is performed through a UART communication interface. With the payload data in hand, the OBC uses the USART interface to transfer it to the UHF transceiver. The UHF transceiver transmits through the UHF antenna TMTC at the frequency of 462 MHz. While the beacons are transmitted by the same antenna at the frequency of 435 MHz. The UHF transceiver is configured for a baud rate of 9600 bits per second.

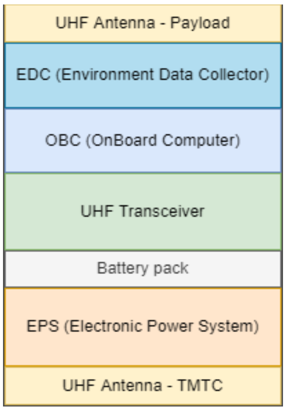

Fig. 4. Hardware architecture block diagram overview

B. *Application of the proposed approach*

Since the EDC software is still under development, the demonstration and evaluation of the proposed approach uses software applications that play the role of the real EDC application payload. The use of different applications has several advantages, particularly as a first evaluation of the proposed approach before using it in the final EDC software still under development. It introduces some diversity to the evaluation since we use three applications and allowed us to select applications with very different profiles, concerning code size, code complexity and, particularly, covering different types and sizes for the data processed by the applications. All the applications were written in C, as this language is highly used in CubeSat software. The applications selected are:

- **Matrices**: it is a program that computes the result of the multiplication of two matrices 30 times and at the end of each run, calculates a cyclic redundancy check (CRC) for the result of the multiplication. After the 30 runs, calculate a final CRC of the 30 CRCs previously calculated. In our experiment, the program uses two 30x30 integer matrices. This is a computation intensive program in a relatively large dataset (due to the 30 iterations of the matrices).
- **PI**: Computes the value of π using the Leibniz formula. In our experiment, the program computed the π using 60000 terms. This is a computation intensive program with a highly sensitive result, as even a minor error in a decimal case of the very long π number calculated will be detected.
- **Fibonacci:** it is a recursive program that computes the sequence of Fibonacci and sums the calculated elements. In our experiment, the program computed the sum of the first 30 elements of the Fibonacci sequence. This is a simple recursive program that intensively uses the stack memory.

It is worth mentioning that these payload software applications (obviously) do not correspond to software payload that is going to fly in future CubeSats from CONASAT-project. However, they represent a variety of application profiles, as mentioned before. Additionally, they are executed in fully realistic conditions (exactly the same conditions as the future EDC software), running on top of the real system software running on the EDC board, namely the FreeRTOS operating system and all the software needed for exchanging messages between EDC board and the OBC board. Furthermore, these applications represent demanding scenarios, as they impose a considerable processing load to the EDC board and reproduce the case of payload software that takes a considerable amount of time to execute (particularly Matrices and PI), which means the successful completion of the application is quite exposed to SEUs.

The fault injection campaigns consist of 10000 faults for the evaluation of the impact of faults in each application. All faults are single bit-flips faults, as this model is widely accepted as a realistic simulation of SEU faults. The target of the injected faults are the registers of the processor of the EDC board. The register affected by each fault was selected randomly (among all the processor registers) and the bit of the register affected by the faults was also selected at random. The trigger of each fault is also randomly defined within an injection window (i.e., within a time interval defined by the tester). The injection window interval was defined as between 2 and 4 seconds after resetting the CubeSat to assure that each fault is injected in the system without having the effects of previous faults. The fault injection (for the software sensitivity evaluation step) was performed using CubeSatFI [39], a fault injection tool that takes advantage of the modern features of the actual microcontrollers by injecting faults in a fully automated way through the JTAG interface. The tool offers the possibility of designing an entire experiment, choosing the specific target that we want to affect, as well as the moment that we want the fault to be injected.

As discussed before, the processor is the main weak point for the reliability of CubeSats boards in the presence of space radiation. In the EDC board, memory is protected with single error correction and double error detection parity codes and the messages exchange between the EDC and OBC boards are also protected with error detection mechanisms.

At the end of the injection window, the target system will be running for a period of 26 seconds to collect data on the impact of the fault for further analysis of the effects of the fault. This period of 26 seconds corresponds roughly to 8 times the average execution time of the application that takes longer, to assure time enough for possible error propagation that may affect the result.

The results produced by each software application are sent to the host computer that controls the experiments through the UART interface of the EDC payload board. The results of the campaigns are saved in a file to further analysis.

These campaigns with randomly injected faults (both in the register space and in time) are appropriated to emulate the effects of transient faults caused by SEU, as space radiation tends to affect the processor in a random way. We decided to keep the single bit-flip model and not to include faults injected in multiple bits of registers because these multiple bits faults (caused by space radiation bursts) tend to cause drastic impact on the software and are easy to detect, and consequently are easy to handle.

It is worth noting that due to the random nature of the injection process, the injected faults may affect either the payload software application or the EDC software, namely the FreeRTOS operating system and the software used for exchanging messages between EDC board and the OBC board, which represents a realistic scenario for SEU faults.

The process was applied following the three additional steps of the proposed approach:

1. Sensitivity evaluation by applying the fault injection campaigns (one campaign for each payload software) to the original software (i.e., without specific SWIFT techniques, unless some error detection techniques such as watchdog timer available in the EDC board).
2. Strengthen of the payload software with a simple SWIFT technique that consists of r-execution of the payload software and voting of the results.
3. Validation of the effectiveness of the SWIFT technique through the fault injection campaigns.

Next subsection presents and discusses the results.

C. *Discussion*

Fig. 5 shows the general impact of faults in the three applications running on the EDC board. It is worth noting that each application was handled independently, as a fault injection campaign of 10000 faults was injected for each application. These experiments correspond to step 1 in a scenario where the EDC application does not have any SWIFT techniques.

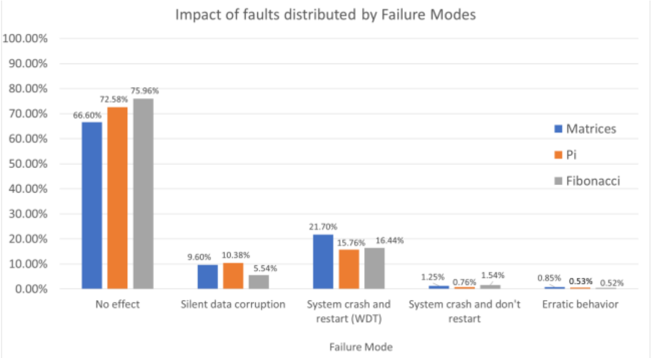

Fig. 5. Impact of faults distributed by failure modes

The failure modes observed are the following:

- **No effect**: The fault had no visible impact on the system, which means that the CubeSat continues to work normally and the expected results are received by the onboard computer.
- **Silent data corruption (SDC)**: The fault had no visible impact on the system. However, the results sent are incorrect. This is the worst type of fault impact, as errors propagated in the system and led to erroneous outputs.
- **System crash and restart (WDT)**: The system crashes and the watchdog timer (WDT) is activated. After the WDT activation, the system is restarted and goes back to working properly.
- **System crash and do not restart**: The system crashes, remaining in that failure mode without WDT activation.
- **Erratic behavior**: The system sends wrong information repeatedly that can be detected by the mechanisms available in the CubeSat.

Considering the failure modes presented above, "silent data corruption (SDC)" is the worst of all and represents a serious risk since the faults that lead to this failure mode are impossible to detect in real situations (in the experiments we can detect these situations because we compared each output with the expected one). The code control flow is not affected, instead, the system produces an erroneous result impossible to be detected, which means that SWIFT techniques must be considered to avoid the serious failure caused by these faults. In contrast, when the system crashes and it is detected by the WDT, the system is restarted and back again to work as expected, ensuring system reliability. Likewise, the "system crash and do not restart" failure mode can be easily managed with external mechanisms such as a Heartbeat system that must receive a signal periodically from the CubeSat payload system. If the signal is not received means that the system is in a blocked situation after a crash and must be forced to restart. Finally, the faults that lead to an "erratic behavior" can be easily detected by the onboard computer that should restart the EDC board.

The high percentage of faults that have no impact ("No effect" failure mode) in three payload application scenarios is quite normal and corroborates previous fault injection experiments reported in the literature (e.g., [12], [13]). This percentage varies between 66% and 76% depending on the software under test. The reason for such results can be justified by the intrinsic redundancy existing in computer systems and software.

Analyzing in more detail the failure mode distribution for the different processor registers, Fig. 6 shows that some processor registers are not affected at all by the injected faults. The reason is because these registers (e.g., R5, R6, R8, R9, R10, R11, R12) are not used by the code. Of course, these situations vary according to the actual software that is being executed in the CubeSat. Furthermore, the way the software uses the available resources of the processor is defined by the C compiler switches during the compilation phase (the EDC system software and all the applications are developed in the C language). In fact, the result of fault injection campaigns can be quite different if the code is compiled with different compilation switches, as this can influence the behavior and performance of the software in execution. Also, the fact that a big number of registers are normally not used by the compiler opens some possibilities to implement extra SWIFT mechanisms.

For page limit reasons, we are not showing the figures equivalent to Fig. 6 for the PI and Fibonacci applications. The results are relatively similar to the ones presented in Fig. 6.

To minimize the impact of the transient faults caused by space radiation, we added the "plain-vanilla" version of the SWIFT technique known as re-execution and voting [36], [37]. In practice, the code is executed twice, and the result is voted in order to decide if it is trustable. If the two results differ, the code is re-executed and compared with the two previous results. In the end, if the third run does not match either of the previous two, a message of error is sent to the output. In contrast, if the result matches one of the first two, it means that one execution was affected by the space radiation but the others can be considered trustable.

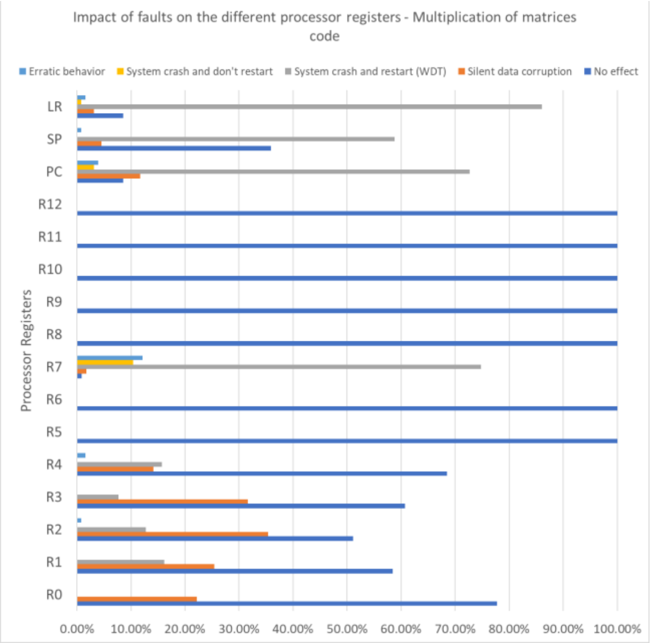

Fig. 6. Impact of faults on the different processor registers - example for the multiplication of matrices application

Fig. 7 shows the distribution of the faults according to the different failure modes in each processor register after the application of the software fault tolerance technique explained above. **The results show that the re-execution and the voter can almost eliminate the impact of faults that cause SDC** on the general registers of the processor that are being used by the code (e.g., R0, R1, R2, R3, R4, R7). On R0 and R7 the silent data corruption is totally tolerated, turning these registers immune to this type of failure mode. Also in the R0, the simple technique of re-executing and voting turns this register immune to space radiation as this register presents a percentage of 100% of "No effect". Positively, in the other registers the percentage of "No effect" added to the percentage of "System crash and restart (WDT", which is also a benign failure mode, is quite close to 100%, showing that the CubeSat software is resilient against faults such as the ones caused by space radiation.

Also in Fig. 7 we can see that the simple software fault tolerance technique used in the multiplication of matrices code just mitigates the effect of faults that leads to SDC on the special registers of the processor (e.g., PC, SP, LR). Looking at the other failure modes, we can see that the results did not change too much for these special registers. This was expected given the relevance of these special registers of the processor, as a fault affecting one of these registers can lead the system to an incoherent behavior or even block the entire system. To strengthen these registers and increase the percentage of "No effect", more sophisticated error detection mechanisms must be added to the software under development (e.g., self-checking routines).

Fig. 8 compares the global impact of the faults considering the three application scenarios before and after the payload applications have been strengthened with the re-execution and voting SWIFT technique. Additionally, the analysis considers only the faults that affected the registers that are being used by the software under test (since faults injected in registers that are not used always lead to "No effect" failure mode).

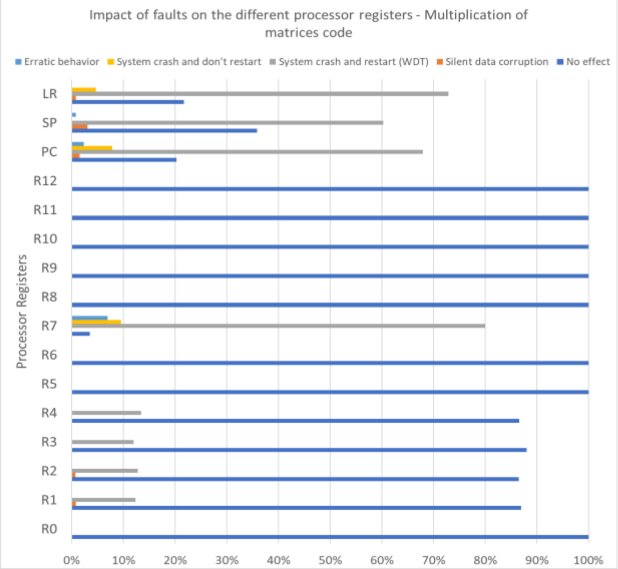

Fig. 7. Impact of faults on the different processor registers - example for the multiplication of matrices application with SWIFI techniques

As mentioned, the impact of transient faults is dependent on the actual software that runs on top of the CubeSat and looking at the two charts in Fig. 8 we can see that the sensibility to space radiation varies according to the software running on the EDC board. The multiplication of matrices is the most sensible code, presenting the lowest percentage of "No effect" (note that we excluded from the analysis the faults that affected registers that are not used by the program, thus all the faults have the potential to affect the program). Although the direct goal of SWIFI techniques is not to increase the percentage of "No effect" failure mode, it is interesting to note that the SWIFI technique actually led to the increase of the percentage of "No effect" in all applications, as a consequence of the masking effect of the SWIFT technique used (i.e., as mentioned above, the code is executed a third time if the results of the two executions are different).

Focusing on the effectiveness of the re-execution and voting in the three software applications, we can conclude that even a simple software fault tolerance technique like this can drastically increase the reliability of the CubeSat. The results presented in Fig. 8 show that the "silent data corruption" failure mode becomes residual, after the introduction of SWIFT technique, and this result is consistent for all the applications. In fact, the percentage of "silent data corruption" the PI application was reduced to 0%, while in the Matrices and Fibonacci was reduced to 0.79% and 0.18%, respectively. With a more sophisticated SWIFT technique (e.g., duplicated voter) the results can be even better.

In addition to the software fault tolerance added to the software, the EDC board includes a watchdog timer (WDT). This error detection mechanism is present in all CubeSats boards and plays a very important role in detecting system crashes. Looking at Fig. 8, we can see that most failures that lead to system crashes are detected by the watchdog timer (i.e., the failures are classified as "System crash and restart (WDT)"). This means that after the system crash is detected by the WDT, the system is restarted and back to work as expected, assuring the availability of the system. Taking advantage of the WDT together with the software fault tolerance technique added to the embedded software, it is possible to make the CubeSats nearly immune to SEU faults.

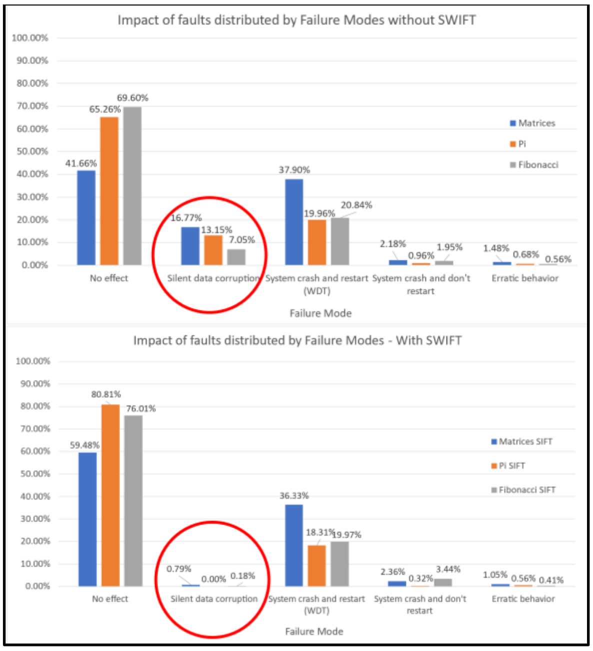

Fig. 8. Comparison of the impact of faults distributed by failure modes on the all the software tested before and after being strengthened with SWIFT techniques

V. Conclusions and Future Work

CubeSats are increasingly popular because of their low cost, as result of the use of commercial off-the-shelf (COTS) components. However, COTS components are susceptible to hardware transient faults caused by space radiation. Hardware fault tolerance in CubeSats is limited to memory and communication protection, leaving the processor unprotected because fault tolerance at processor level would dramatically increase CubeSats cost, energy consumption, and weight (hurting the three capital advantages of CubeSats: low cost, low energy needs, and low weight). In practice, the situation is that CubeSat cannot tolerate the faults cause by radiation and are still regarded as satellites that cannot be used in serious and critical missions.

The obvious solution of tolerating hardware transient faults caused by radiation at the software level is also not used in real CubeSats (although there are a couple of proposals in the literature), mainly because the systematic use of a software fault tolerance architecture covering system software (e.g., operating systems) and applications would make the software project of CubeSats hugely complex, particularly if the fault tolerance techniques are transparent for the developer of CubeSats software applications.

This paper proposes a set of additional steps to the software development process used in the development of CubeSats software applications with the goal of making CubeSats immune to space radiation faults. The proposed solution intends to be easy to adopt to the software development life cycle used by companies, space agencies, and other institutions that are developing CubeSats. The key idea is to use two well-known ingredients (fault injection and SWIFT techniques) in a way that is compatible with the known budget restrictions of CubeSats.

The proposal adds the systematic measurement of the CubeSat software sensitivity to transient faults caused by space radiation as a mandatory step of the software development process, as part of the V&V stages. Depending on the results of the sensitivity analysis, the proposed approach recommends the enhancement of the software under development with targeted SWIFT techniques, to make the software resilient to the transient faults. The novelty is in the fact that the proposed SWIFT techniques are targeted (i.e., just enough and as simple as possible) and are applied only at the software application level. Although this means that the SWIFT techniques are not transparent for the developers (i.e., represent an extra effort), this approach keeps the simplicity required by CubeSats projects.

The evaluation of the software sensitivity to transient faults can be easily achieved using affordable fault injection. Since it is well-documented in the literature that the impact of transient faults is highly dependent on the actual software code, general and very expensive software fault tolerant architectures (which are not yet available for CubeSats) are not recommended because of two reasons: **a)** they may be not needed in many cases (as the inherent redundancy of software tolerates the fault, as shown by the high percentage of "no effect" faults) and **b)** they impose a fixed and very high cost and complexity to the CubeSat software projects. These two reasons support our proposed approach of just enough SWIFT techniques to avoid silent data corruption failures, which are the real problematic impact of space radiation.

In short, we can summarize the proposed solution in the following steps: 1) Evaluation of the software sensitivity to space radiation; 2) Strengthen the software with tailored software implemented fault tolerance (SWIFT) techniques; 3) Validate the effectiveness of the resulting software with SWIFT techniques. These 3 steps iterate until the required software resilience is achieved.

The proposed approach is evaluated using a real CubeSat board and three different software applications specifically designed to cover different features such as complexity, data size and execution type (iterative and recursive.) The results of injecting 30 thousand faults show that it is possible to reduce the percentage of faults that caused silent data corruption to residual values (or even to 0% in one of the applications used in the experiments, which means that the CubeSat application became immune to faults such as the ones caused by space radiation for the critical silent data corruption failure mode. Furthermore, the SWIFT technique used in the experiments consisted of simple software component re-execution and voting, which is a simple technique that can be easily added to software applications.

Future work includes the evaluation of the real EDC application software that is currently under development and consequent improvement of the software with targeted SWIFT techniques to make it immune to space radiation as much as possible. The operating system used in these particular CubeSat boards (FreeRTOS) can also be enhanced with several mechanisms and new system calls to facilitate the development of SWIFT techniques at the application software level. Additionally, the development of a repository (library) of reusable SWIFT techniques for CubeSat software is also being considered, as this will reduce the effort of developing fault tolerance at the application level.

Acknowledgment

This work was supported in part by the Grant CISUC-UID/CEC/00326/2020, funded in part by the European Social Fund, through the Regional Operational Program Centro 2020, and supported in part by the H2020, Marie-Curie project ADVANCE ("Addressing Verification and Validation Challenges in Future Cyber-Physical Systems"), project funded by the VALU3S ("Verification and Validation of Automated Systems' Safety and Security") project, supported by the European Commission, and in part by European Leadership (ECSEL) Joint Undertaking (JU) under Grant 876852, and in part by the JU from the European Union's Horizon 2020 Research and Innovation Programme.

REFERENCES

[1] "CubeSat Design Specification (1U - 12U) REV 14", CP-CDS-R14.

[2] R. Ecoffet, "Spacecraft Anomalies Associated with Radiation Effects", in RADECS 2013 Short Course Proceedings, Chap. VIII, 2013.

[3] F. Davoli, C. Kourogiorgas, M. Marchese, A. Panagopoulos, F. Patrone, "Small satellites and CubeSats: Survey of structures, architectures, and protocols", Int. Journal of Satellite Communications and Networking, September 2018.

[4] T. K. Moon, "Error Correction Coding. New Jersey: John Wiley & Sons", ISBN 978-0-471-64800-0, 2005.

[5] Z. Yuan and X. Zhao, "Introduction of forward error correction and its application," 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), 2012.

[6] D. Sorin, "Fault tolerant computer architecture." *Synthesis Lectures* on Computer Architecture 4.1 (2009): 1-104. 2009.

[7] C. M. Fuchs, "Fault-tolerant satellite computing with modern semiconductors," Ph.D. dissertation, Leiden University, 2019.

[8] M. Langer and J. Bouwmeester, "Reliability of CubeSats-statistical data, developers' beliefs and the way forward," in AIAA/USU Conference on Small Satellites (SmallSat), 2016.

[9] M-C Hsueh, T. K. Tsai and R. K. Iyer, "Fault injection techniques and tools," in Computer, vol. 30, no. 4, pp. 75-82, April 1997.

[10] R. Natella, D. Cotroneo, H. Madeira, "Assessing Dependability with Software Fault Injection: A Survey", ACM Computing Surveys 48 (3), 2016.

[11] R. Barbosa, N. Silva, J. Duraes, H. Madeira, "Verification and validation of (real time) COTS products using fault injection techniques", COTS-Based Software Systems, ICCBSS'07, 39, 2007.

[12] H. Madeira, R.Some, F. Moreira, D. Costa, D. Rennels, "Experimental evaluation of a COTS system for space applications", Int. Conference on Dependable Systems and Networks, DSN-2002, USA, 2002.

[13] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors in Programs," IEEE Transactions on Dependable and Secure Computing, 2020.

[14] R. Twiggs, "Origin of cubesat," Small Satellites: Past, Present, Future, Eds: Helvajian H., Janson SW, The Aerosp Press, California, 2008

[15] C. Batista, A. Weller, E. Martins, and F. Mattiello-Francisco, "Towards increasing nanosatellite subsystem robustness," Acta Astronautica, vol. 156, pp. 187–196, 2019

[16] D. Almeida and F. Mattiello-Francisco, "Modeling of the interoperability between on-board computer and payloads of the nanosat-br2 with support of the uppaal tool," in 1st IAA Latin American Symp. on Small Satellites. Colombia, 2017

[17] C. Batista, T. Basso, F. Mattiello-Francisco and R. Moraes, "Impacts of the Space Technology Evolution in the V&V of Embedded Software-Intensive Systems" in The 2020 International Conference on Computational Science and Computational Intelligence (CSCI´20: December 16-18, USA, 2020.

[18] S. A. Jacklin, "Survey of Verification and Validation Techniques for Small Satellite Software Development", NASA Ames Research Center, 2015 Space Tech Expo Conference, May 19-21, 2015.

[19] H. P. Zima, M. L. James, and P. L. Springer, "Fault-tolerant on-board computing for robotic space missions," Concurrency and Computation: Practice and Experience, vol. 23, no. 17, pp. 2192–2204, Dec. 2011.

[20] D. Briere and P. Traverse, "Airbus A320/A330/340 electrical flight controls a family of fault-tolerant systems," Digest of Papers - International Symposium on Fault-Tolerant Computing, pp. 616–623, 1993.

[21] Y. C. Yeh, "Safety critical avionics for the 777 primary flight controls system," AIAA/IEEE Digital Avionics Systems Conference - Proceedings, vol. 1, 2001, doi: 10.1109/DASC.2001.963311.

[22] C. H. Stapper, V. K. Jain, and V. K. Jain, Defect and Fault Tolerance in VLSI Systems, 1st ed., vol. 2. New York, NY: Springer, 1990.

[23] T. C. Bressoud, "TFT: A software system for application-transparent fault tolerance," Digest of Papers - 28th Annual International Symposium on Fault-Tolerant Computing, FTCS 1998, vol. 1998-January, pp. 128–137, 1998, doi: 10.1109/FTCS.1998.689462.

[24] B. Hasircioglu, Y.-A. Pignolet, and T. Sivanthi, "Transparent Fault Tolerance for Real-Time Automation Systems," Proceedings of the 1st International Workshop on Internet of People, Assistive Robots and Things, pp. 7–12, 2018, doi: 10.1145/3215525.3215538.

[25] L. L. Pullum, "Software Fault Tolerance Techniques and Implementation". USA: Artech House, Inc., 2001.

[26] M. Yang, G. Hua, Y. Feng, and J. Gong, "Fault-Tolerance Techniques for Spacecraft Control Computers". 2017.

[27] S. Mukherjee, "Architecture Design for Soft Errors", ed. by S. Mukherjee, Morgan Kaufmann, 2008.

[28] N. Murphy, "Watchdog Timers", in Embedded Systems Programming, 2000.

[29] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 9, 6, 1990.

[30] J. Vankeirsbilck, N. Penneman, H. Hallez, and J. Boydens, "Random additive signature monitoring for control flow error detection," IEEE Transactions on Reliability, vol. 66, no. 4, pp. 1178–1192, Dec. 2017.

[31] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," IEEE Transactions on Software Engineering, vol. 24, no. 2, 1998.

[32] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic object-oriented fault injection tool," Proceedings of the International Conference on Dependable Systems and Networks, pp. 83–88, 2001.

[33] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann and O. Spinczyk, "FAIL: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," *2015 11th* European Dependable Computing Conference (EDCC), 2015.

[34] L. Feinbube, L. Pirl and A. Polze, "Software Fault Injection: A Practical Perspective", book chapter, in "Dependability Engineering", Ed. F. García Márquez and M. Papaelias, ISBN 978-1-78923-259-2, 2018.

[35] Heinig, A., Korb, I., Schmoll, F., Marwedel, P. & Engel, M., "Fast and low-cost instruction-aware fault injection", In Horbach, M. (Hrsg.), INFORMATIK 2013.

[36] I. Koren and C. Krishna, "Fault-Tolerant Systems", Elsevier, 2nd Edition – Sept. 2020.

[37] M. R. Lyu, "Software Fault Tolerance", John Wiley & Sons Ltd, 1st Ed 1995.

[38] Horst Schirmeier, "Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance", PhD thesis, University of Dortmund, 2016.

[39] D. Paiva, J. M. Duarte, R. Lima, M. Carvalho, F. Mattiello-Francisco and H. Madeira, "Fault injection platform for affordable verification and validation of CubeSats software", 10th Latin-American Symp. on Dependable Computing (LADC), 2021.

[40] INPE. "Environmental Data Collector (EDC)". INPE, July 5, 2021. http://www.inpe.br/crn/projetos/edc.php Accessed on: September 23, 2022.

[41] K. P. Queiroz, S. M. Dias, J. M. Duarte, M. M. Carvalho, "Uma Solução Para O Sistema Brasileiro De Coleta De Dados Ambientais Baseada Em Nanossatélites", Holos, Dez, 2018.