On the use of a Failure Emulator Mechanism at Nanosatellite Subsystems Integration Tests

Carlos L. G. Batista
Space Systems Engineering
National Institute for Space Researches
São José dos Campos, SP, Brazil
Email: carlos.gomes@crn.inpe.br

Eliane Martins
Institute of Computing
Campinas University
Campinas, SP, Brazil
Email: eliane@ic.unicamp.br

Maria de Fátima Mattiello-Francisco
Space Systems Engineering
National Institute for Space Researches
São José dos Campos, SP, Brazil
Email: fatima.mattiello@inpe.br

***Abstract*—The increase number of CubeSat based space missions in the last decade shows the new possibilities for cheaper, faster but not so better projects. This lack of quality in a mission completeness point of view is associated to lack of good practices at developing, assembly and testing phases. Addressing such problem, this work presents a fault injection tool for nanosatellite robustness testing, named Failure Emulator Mechanism (FEM). The goal of the FEM is to emulate at the subsystems interface level faults that could be presented during the mission operation of the spacecraft. Working at the communication bus, the FEM is capable to intercept the exchanged messages between two subsystems under test and inject different faults: (i) time related faults, i.e. delay; (ii) value related faults, i.e. bitflip and; (iii) specific communication bus faults, i.e. a verbose subsystem. The FEM was developed to support the integration tests of the software-intensive NanoSatC-BR2 On Board Data Handling Software (OBSw) and its Payloads. The NanoSatC-BR2 is the second scientific nanosatellite developed jointly by the Brazilian National Institute for Space Researches (INPE) and Santa Maria Federal University (UFSM). As this spacecraft works with a full shared I2C communication bus, the FEM was implemented to support and work at this communication protocol and electric interface. The use of FEM has proved to be helpful along all phases of nanosatellite development. In the early phase, FEM supports robustness requirement validation by means models in the loop (MIL). In the nanosatellite integration phase, FEM supports robustness testing of the communicating subsystems under integration, configuring hardware in the loop (HIL). In this paper, we present the design, implementation and results of FEM as MIL tool for robustness requirement validation of OBSw and Langmuir Probe, a particular NanoSatC-BR2 Payload.**

***Index Terms*—Fault Injection, Nanosatellites, CubeSat, testing, MIL, robustness.**

I. Introduction

The nanosatellite cubesat-based missions have become more and more commercially attractive due to the low mission development cost, being a great opportunity to test new technologies at the hazardous space environment [1], [2], [3], [4], [5], [6].

The main low-cost reasons are the standardization of Cubesat platform [7], [8] and the way nanosatellites are placed in orbit, being pigback at big space vehicles. The former contributes to reducing project lifetime, saving time for platform subsystem design and testing; the last contributes to share launch cost. However, studies have shown high mission failure rate due to lack of rigorous development and verification system [9].

Best practices for CubeSats development have been proposed in the last years to improve the success rates of these missions. Examples at Systems Engineering try to create and evaluate different approaches for the concept and development of Cubesat-based missions [10], [11].

Rather than increasing quality assurance along the whole mission lifecycle, which will make the mission unfeasible under cost perspective, our approach focuses on the software-intensive mission payload only. Assuming the payload is usually new design to be integrated in the Cubesat standard, much attention on it must be paid in order to assure the robustness of other subsystems in the interaction with it, in particular the On-Board Datahandling Software (OBSw).

Robustness is the system property of dealing with unexpected event at certain operational situations without affecting the service provided [12]. Better robustness requirements lead to better models development that leads to better software implementations and, at the end, more reliable systems.

This work present a fault injection tool to support the development, integration and tests of a given subsystem payload of a cubesat-based nanosatellite mission, focusing on the subsystems robustness verification at interface level.

The Section II shows the interface injection approach and its two implementation possibilities (Subsections II-A and II-B). In the next Section, III it is presented the Failure Emulator Mechanism, objective of this paper, and its principal characteristics (Subsections III-A and III-B). This paper ends with the environment for tests on the Section IV and finally its Conclusions about the tool capabilities and future work.

II. Interface Fault Injection

The interface fault injection is an approach for Software Implemented Fault Injection – SWFI – which consists of insert faults at a specific interface between two communicating systems [13].

At this level is possible to inject faults and emulate failures on the messages exchanged between the two or more systems. Commonly faults inserted at the interface layer are faults related to robustness, e.g. change expected values and messages, and the implemented communication protocol itself, e.g. change electric signals.

There are two ways to implement interface fault injectors, using a driver or a interceptor, see Figures 1 and 2.

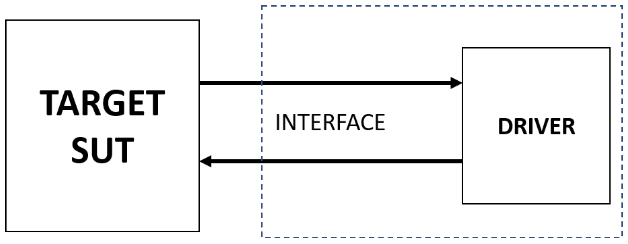

Fig. 1. Driver Fault Injector Implementation

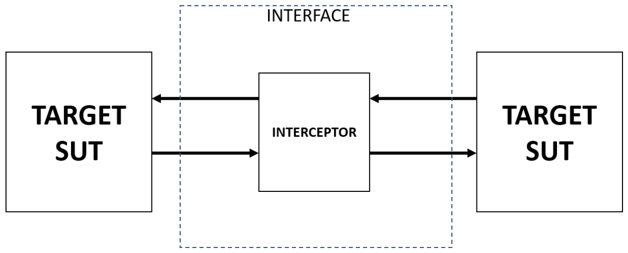

Fig. 2. Interceptor Fault Injector Implementation

A. Driver Fault Injector

This implementation consists on a ”dummy” system connected to the target SUT emulating the behavior of another communicating system. The role of this driver is to exchange messages with the target System Under Test – SUT – like a real system and inject possible faults from the systems connected on chosen interface.

This implementation is normally used to simulate the environment and test one target system in a hardware in the loop approach. Acceptance test phases normally use this kind of test driver in a way to evaluate functional characteristics of the target SUT.

B. Interceptor Fault Injector

The interceptor implementation consists on monitoring the exchanged messages between the two target SUT and insert or not faults at the communication channel.

The role of the interceptor is to emulate a faulty communication channel, intercept the messages with the minimum interference and inject faults, if necessary.

This implementation can be used at integration tests without need to know the exactly behaviour of the target systems, ”black box” tests. But it requires a well-defined and well-know interface and communication protocol to keep low interference delays and enable the fault injector to interoperate properly with the SUT.

Previous works of fault injection on interface level [14], [15], [16] and robustness tests [17] prove the utility of this approach and the NASA Software Safety Guide Book (NASA-GB-8719.3) recommends fault injection for assessing the system behaviour against faulty third party components (e.g. COTS), provoking transitions that are not allowed by the requirements [18].

III. Failure Emulator Mechanism

The Failure Emulator Mechanism – FEM – is an interface fault injector based on the interceptor implementation. It is idealized as a robustness testing tool to support the development and integration of nanosatellite subsystems, in special the spacecraft On Board Computer – OBC – and its Payloads – PLD.

This tool is capable of intercepting all the messages exchanged between the OBC and one of its PLD, emulating failures through the injection of controlled faults at communication level based on a criteria [19].

For this role, the FEM was implemented as a device that can take the role of a slave device when communicating with the OBC (original master device) and take the master role from the PLD (original slave device) point of view.

The Figure 3 represents the architecture for the FEM implementation.

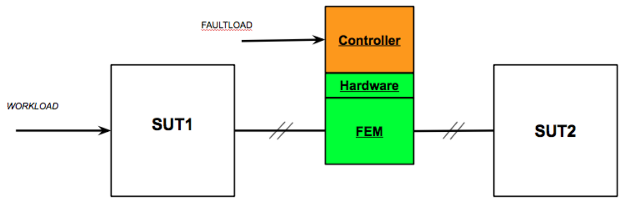

Fig. 3. FEM Architecture

To a proper functionality of the interceptor, two items must be determined: the interface (logic and electric) and the faultload (the possible faults to be inject and the script for FEM actuation).

A. The Interface

As an interceptor fault injector, the FEM needs a well-known interface description at electric and logic levels.

As described, the interface between the SUT is the I^2C, communication protocol and electric interface. The I^2C was created by Philips as Inter-IC, a fast and simple communication protocol for Integrated Circuits with the minimum use of pins [20].

This protocol is being used by the great majority of the CubeSat mission due to its adoption by the commercial nanosatellites developers around the world and easily implementation on COTS microcontrollers. The plug-and-play strategy of the most of the CubeSat-based missions made the I^2C the first choice for the industry as communication bus. Even with new projects, using SPI, UART or CAN, major nanosatellite enterprises and universities projects still use the Phillip’s protocol as a simple way to achieve they objective.

The I^2C works as a Master-asks/Slave-responds protocol with broadcast messages. Only two bus lines are required, a serial data line (SDA) and a serial clock line (SCL), and all the devices are identified by a unique 7-bit address.

The Master device starts all the communications and controls the messages synchronisation. The Figure 4 shows the common I^2C frame structure. All messages have the same structure:

```
START BIT + SLAVE ADDRESS +..
..+ READ/WRITE BIT + ACKNOWLEDGE +..
..+ DATA + ACKNOWLEDGE +..
..+ DATA + ACKNOWLEDGE +..
..+ .. + ACKNOWLEDGE + STOP BIT
```

The electric interface for I^2C communication is a two wire interface TWI – one for the SDA and another for the SCL. Both are pulled-up to the logic high level by resistors connected to a single positive supply, usually +3.3 V or +5 V. All connected devices have open-collector driver stages that can transmit by pulling the bus low, and high impedance sense amplifiers that monitor the bus voltage to receive data.

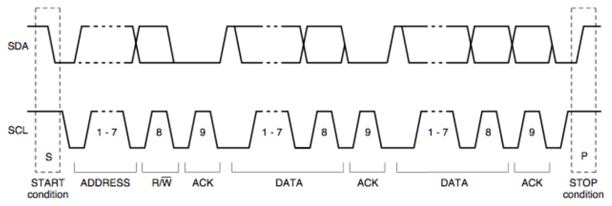

Fig. 4. I^2C Frame Structure

B. The Faultload

The Faultload is the FEM main conductor during the tests. This role is performed by two files uploaded to an SD card where the FEM downloads and stores the information of these files as an guide to how will be its behaviour during each test phase.

The two files are (i) a configuration file (`CONFIG.TXT`) with the identification of the test case, the systems under test (the I^2C addresses in decimal representation of each SUT) and the number of faults to inject; and (ii) a faultload script file (`FEM.TXT`) with information about the faults to inject, i.e. where to inject, when to inject, what to inject and how to inject.

Both files are an array of *integers*, the first one is a 1×4 array and the second one, a $n \times 4$ array, where the n is the number of faults to inject captured from the configuration file.

```
int array CONFIG[id, SUT1, SUT2,..
    ..nbr-of-faults];

int array FEM[where, when, what, how];
```

The faults chosen are based on emulate the degradation of the service delivered, early study [21] defined on fault injection to suppose three major types of faults: as value faults (the transmission of incorrect values), provision faults (transmission of unexpected information) and time faults (time related faults).

The next points present a brief explanation about each element of the FEM's array in `FEM.TXT` file.

1) Where: Defines where a fault will be inject. For the I^2C protocol, on a READ or on a WRITE message. This column could have two values: `0x00 (WRITE)` or `0x01 (READ)`.

```
enum WHERE{
    READ    = 0x00,
    WRITE   = 0x01
}
```

2) When: Defines the exact instant to inject a fault, could be in time or counting the number messages exchanged. For the I^2C protocol it is used the number of messages, `READ` or `WRITE`.

```
int WHEN;
```

3) What: Defines the fault to be inject. For this implementation it is used four different fault types: (i) value fault, implemented through a bitflip fault, (ii) provision fault, implemented changing the message information, (iii) time fault, insertion of delays on exchanged messages, and (iv) protocol specific fault, for this implementation lock down the data I^2C bus line. Each fault is identified at the array as an integer, as demonstrated, the addition of new faults must be followed by an unique identification too.

```
enum WHAT{
    BITFLIP = 0x01,
    DELAY     = 0x03,
    CHANGE  = 0x05,
    LOCK    = 0x07
}
```

4) How: Defines the argument for each implemented fault, i.e. which bit to flip on a bitflip fault, how many milliseconds to delay on time faults, the new byte to transmit on a provision fault, how many milliseconds to lock down the data bus.

```
int HOW;
```

IV. The MIL Environment

To validate the capabilities of FEM it is necessary to insert it in a controlled environment with two well-known SUTs.

For that, it is used a Model-in-the-Loop – MIL – environment with the OBC and one of its PLD modeled as describe by the Spacecraft Interface Control Document [22]. This document is part of the NanoSatC-BR2 mission documents.

The NanoSatC-BR2 mission is the second spacecraft of the scientific nanosatellites concepted by the Brazilian National Institute for Space Researches INPE with the objective to study the Earth Magnetosphere.

In this paper we present the use of FEM on the verification by testing robustness properties of the OBSw embedded in OBC when communicating with a particular NanoSatC-BR2 payload, named Langmuir Probe (SLP), which is also software intensive subsystem. The expected interaction of these two SUT is presented at the sequence diagram on Figure 5.

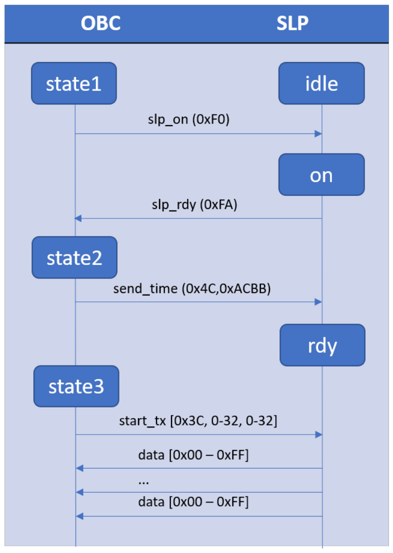

Fig. 5. OBC-SLP Interface Sequence Diagram

The communication starts with a handshake between the two systems, with the OBSw sending a *slp_on* to wake up the SLP who answer with a *slp_rdy* to indicate that it is ready to continue. The next thing to do is to set up the the initial time, *send_time* for the data recording with the OBSw sending a timestamp, here a two byte data `<0xACBB>`. From now the OBSw consider the SLP ready to start the data acquisition and begin the transmission as soon it is called again. With the *start_tx* command followed by the start and end pointers for the SLP *32-byte-register* the SLP is allowed to start the transmission of the acquired data to the OBSw, byte-at-byte (`0x00-0xFF`)

All the MIL environment is supported by Arduino Uno boards. This choice was made looking for cheap and easy solutions for development environment as part of the COTS (Commercial Off the Shelf) philosophy attached to the nanosatellites projects.

The use of these kind of boards was at the beginning a problem, once the Uno boards have only one TWI ports. Looking for a solution it is used an open-source library `<I2CSoftMaster.h>`, library to transform two other Uno digital pins into TWI pins.

With this problem solved the Figure 6 shows the FEM class diagram implemented at the final version.

The Table I shows the information of the configuration and faultload files representing a particular test case generated to validade the FEM capabilities during a test at the OBSw and SLP model integration level.

All the faults were software implemented as described, the exception, the lock fault injection. Once this is a physical fault, it is necessary to realy force the SDA bus down and keep it down for how long it is necessary. For that it is used one of the Arduino Digital pins (e.g. `pin 8`) connect to the base

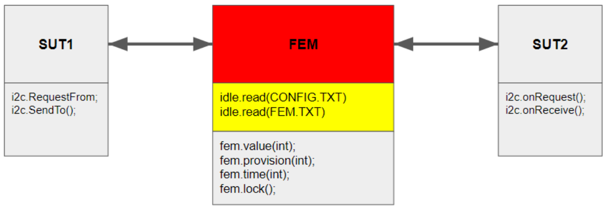

Fig. 6. FEM Class Diagram

of a bipolar junction transistor, in a ”NOT GATE” connection logic, with the collector with the SDA bus (between the master device and FEM) and the emitter grounded.

TABLE I
TEST CASE TO VALIDADE THE USE OF FEM TESTING THE OBC AND SLP

ID	SUT1	SUT2	NFAULTS
test-case02	OBSw	SLP	10 faults

WHERE	WHEN	WHAT	HOW
write	9	*DELAY*	5 ms
read	13	*LOCK*	100 ms
write	21	*DELAY*	5 ms
read	25	*LOCK*	100 ms
read	29	*LOCK*	100 ms
write	33	*BITFLIP*	LSB 2
write	37	*DELAY*	5 ms
read	41	*CHANGE*	0x02
write	45	*DELAY*	5 ms
write	49	*DELAY*	5 ms

This test case was uploaded to the Arduino board through the SD card. The selected values for each fault parameter was chosen to confirm consistance at the results, i.e. the LOCK fault to supress the messages ($100ms$ is enough to perform a supression at master point of view), $5ms$ for the delay, change to `0x02` and bitflip at `LSB 2`.

The Tables II and III show the tests done with the FEM to analyse its behaviour and intrusion at the SUT communication, respectively on `WRITE` and `READ` demands.

TABLE II
SUMMARY OF FEM FUNCTIONS UNDER TESTS ON WRITE MESSAGES

MASTER Tx	FAULT	PARAM	DELAY	SLAVE Rx
0xF0	null	N/A	758 us	F0
0x3C	delay	5 ms	5628 us	FF
0x3C	delay	20 ms	5598 us	FF
0xF0	bitflip	lsb 2	1543 us	0xFB
0xF0	delay	5ms	5543 us	FF
0x3C	delay	5 ms	5601 us	FF
0x3C	delay	5 ms	5599	FF

The results presented in both tables (II,III) were observed by Serial Monitor and using an oscilloscope with I^2C digital decoder as show in Figure 7 which demonstrates a value (bit-flip) fault injected at the attempt to `WRITE 0xF0` and it

TABLE III
SUMMARY OF FEM FUNCTIONS UNDER TESTS ON READ MESSAGES

SLAVE Tx	FAULT	PARAM	DELAY	MASTER Rx
0xFA	null	N/A	1061 us	FA
0xFA	delay	5 ms	6744 us	0xFF
0xFA	lock	100 ms	101 ms	0xFF
0xFA	lock	100 ms	101 ms	0xFF
0xFA	change	0x02	1077 us	0x02

becames `0xFB` due to a the bit-flip at the `LSB 2`, third bit at the byte transmitted.

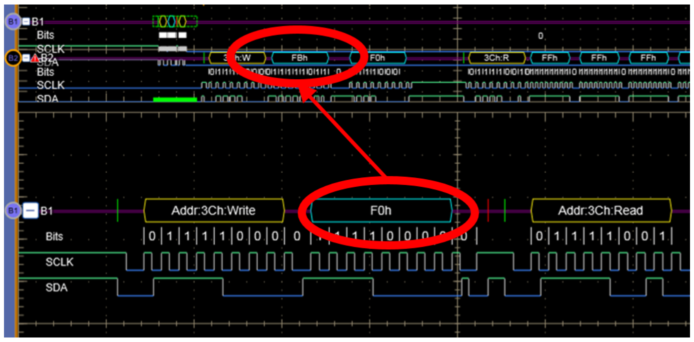

Fig. 7. Example of a VALUE fault injection during a WRITE transmission

V. CONCLUSION

The evaluation of robustness through testing software-intensive systems is crucial for the success of space missions. The complete operation of the mission depends on many different components/subsystems operating correctly and delivering the correct services. So, it is a good practice to anticipating verification and validation of functional and robustness requirements of communicating software-intensive subsystems as early as possible besides the environmental and launch vehicle constraints only.

The FEM shows itself as an impressive tool to support the robustness tests even in a MIL environment, less intrusive than expected on the I^2C protocol. Its capabilities, for emulating by fault injection possible non-expected situations, support the integration tests and exploit the development phase helping the subsystem team to foresee events and reduce the rework.

The low memory of the Arduino Uno is one of the negative points for this implementation and also its low processing power, even not being responsible for big troubles. Future works with other communication protocol may need more processing budget.

The next steps of this work are: (i) continue with the second phase using HIL and test the final implementations of the OBC and NanosatC-BR2 payloads, (ii) exploit the use of the FEM in a test chain where we can model the SUTs, automatic generate test cases with faultload, inject the faults and get logs to feedback the testers and, (iii) develop a FEM framework to implement new solutions to fault injection and robustness testing using different space communication protocols (CAN, SPI, UART) and different boards, like Arduino Mega, Blue Pill, STM, even Raspeberry Pi to give the FEM more capabilities.

A satellite, even a nano-one, is not all software-intensive and the FEM demonstrates here one of its advantages. The use of a model approach on the FEM implementation allows hardware solutions to inject faults, e.g. lock down the I^2C bus through a transistor in an open-collector configuration. Exploiting the other satellite interfaces and studying more hardware fault effects on the system as whole could be a way to put the FEM as a mid-term between software and hardware injector.

ACKNOWLEDGMENT

The authors would like to thank the CAPES for the scholarships which enable the authors to acomplish this work, the University of Campinas, INPE and all involved on this paper.

REFERENCES

[1] D. Selva and D. Krejci, "A survey and assessment of the capabilities of Cubesats for Earth observation," *Acta Astronautica*, vol. 74, pp. 50–68, 2012. [Online]. Available: https://pdfs.semanticscholar.org/ee1e/.../b3cd.pdf

[2] G. Gustavo, C. Mariño, E. E. Bürger, G. Loureiro, and O. L. Bogossian, "IAA-BR-16-1P-01 Mission analysis for a remote sensing CubeSat mission over the Amazon rainforest," 2012. [Online]. Available: http://mtc-m21b.sid.inpe.br/col/sid.inpe.br/mtc-m21b/2016/03.03.12.31/doc/coronel{_}mission.pdf

[3] A. Poghosyan and A. Golkar, "CubeSat evolution: Analyzing CubeSat capabilities for conducting science missions," *Progress in Aerospace Sciences*, vol. 88, pp. 59–83, 2016.

[4] Space Studies Board, Division on Engineering and Physical Sciences, and National Academies of Sciences Engineering and Medicine, "Achieving Science with CubeSats: Thinking Inside the Box," *Nap 23503*, no. September, p. 130, 2016. [Online]. Available: http://www.nap.edu/catalog/23503

[5] S. Nag, "Sensor Webs of Agile, Small Satellite Constellations and Unmanned Aerial Vehicles with Satellite-to-Air Communication Links," in *1st IAA Latin American Symposium on Small Satellites*, 2017. [Online]. Available: http://www.unsam.edu.ar/.../Presentaciones/Session.5/IAA-LA-05-05.pdf

[6] D. F. M. Mota, "OPENOBC: uma arquitetura de um computador de bordo open source e de baixo custo para o padrão CUBESAT," Ph.D. dissertation, Universidade Federal do Ceará, jan 2017. [Online]. Available: http://repositorio.ufc.br/ri/handle/riufc/22986

[7] CalPoly, "Cubesat design specification rev. 13," *The CubeSat Program, California Polytechnic State ...*, vol. 8651, p. 22, 2009. [Online]. Available: http://wdww.cubesat.org/images/developers/cds{_}rev13{_}final2.pdf

[8] J. Puig-Suari, C. Turner, and R. Twiggs, "CubeSat: the development and launch support infrastructure for eighteen different satellite customers on one launch," *15TH Annual/USU Conference on Small Satellites*, pp. 1–5, 2001. [Online]. Available: http://digitalcommons.usu.edu/smallsat/2001/All2001/59/

[9] M. Swartwout, "Secondary spacecraft in 2016: Why some succeed (And too many do not)," in *2016 IEEE Aerospace Conference*. IEEE, mar 2016, pp. 1–13. [Online]. Available: http://ieeexplore.ieee.org/document/7500791/

[10] D. Kaslow, B. Ayres, P. T. Cahill, L. Hart, and R. Yntema, "A Model-Based Systems Engineering (MBSE) approach for defining the behaviors of CubeSats," in *2017 IEEE Aerospace Conference*. IEEE, mar 2017, pp. 1–14. [Online]. Available: http://ieeexplore.ieee.org/document/7943865/

[11] D. Kaslow, L. Anderson, C. Iwata, S. Asundi, B. Ayres, and R. Thompson, "Developing and Distributing a CubeSat Model-Based Systems Engineering (MBSE) Reference Model," in *Proceedings of the 31st Space Symposium*, 2015, pp. 1–14. [Online]. Available: http://2015.spacesymposium.org/sites/default/files/downloads/

[12] J. Radatz, A. Geraci, and F. Katki, "Ieee standard glossary of software engineering terminology," *IEEE Std*, vol. 610121990, no. 121990, p. 3, 1990.

[13] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing Dependability with Software Fault Injection : A Survey," *ACM Computing Surveys*, vol. 48, no. 3, pp. 44:1–44:55, 2016. [Online]. Available: http://dx.doi.org/10.1145/2841425http://dl.acm.org/citation.cfm?id=2841425

[14] R. J. Dobler, S. Cechin, T. Weber, and J. Netto, "A Software Fault Injector to Validate Implementations of a Safety Communication Protocol," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. IEEE, oct 2016, pp. 35–42. [Online]. Available: http://ieeexplore.ieee.org/document/7781834/

[15] M. d. F. Mattiello-Francisco, "Inrob–uma abordagem para teste de interoperabilidade e de robustez de subsistemas de tempo-real intensivos em software," Ph.D. dissertation, PhD Thesis. Aeronautical Institute of Technology, Brazil, 2009.

[16] A. C. Weller, E. Martins, and F. Mattiello-Francisco, "Inrob-uml: uma abordagem para testes de interoperabilidade e robustez baseados em modelos," *SAST 2015*, p. 71, 2015.

[17] A. K. Ghosh, M. Schmid, and V. Shah, "Testing the robustness of windows nt software," in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*. IEEE, 1998, pp. 231–235.

[18] D. Cotroneo and R. Natella, "Fault injection for software certification," *IEEE Security & Privacy*, vol. 11, no. 4, pp. 38–45, 2013.

[19] A. Johansson and N. Suri, "Error Propagation Profiling of Operating Systems," *Proc. IEEE/IFIP Intl. Conf. Dependable Systems and Networks*, 2005. [Online]. Available: http://research.cs.wisc.edu/areas/os/ReadingGroup/OS/papers/errprop-dsn05.pdf

[20] N. Semiconductors, "UM10204 I2C-bus specification and user manual," *Semiconductors*, vol. 3, no. June, pp. 1–50, 2014.

[21] V. Bonfiglio, L. Montecchi, I. Irrera, F. Rossi, P. Lollini, and A. Bondavalli, "Software faults emulation at model-level: Towards automated software fmea," in *Dependable Systems and Networks Workshops (DSN-W), 2015 IEEE International Conference on*. IEEE, 2015, pp. 133–140.

[22] D. Palamin, "Modelling of the interoperability between on-board computer and payloads of the nanosatc-br2 with support of the uppaal tool," 2017, 1st IAA Latin American Symposium on Small Satellites. [Online]. Available: http://www.unsam.edu.ar/institutos/Colomb/Presentaciones/Session.9/IAA-LA-09-01.pdf