Toward Applying Fuzz Testing Techniques on the SUCHAI Nanosatellites Flight Software

1st Tamara Gutierrez
Department of Computer Science
University of Chile
Santiago, Chile
tamara.gutierrez@ug.uchile.cl

2nd Alexandre Bergel
Department of Computer Science
University of Chile
Santiago, Chile
abergel@dcc.uchile.cl

3rd Carlos E. Gonzalez
Electrical Engineering Department
University of Chile
Santiago, Chile
carlgonz@uchile.cl

4th Camilo J. Rojas
Electrical Engineering Department
University of Chile
Santiago, Chile
camrojas@uchile.cl

5th Marcos A. Diaz
Electrical Engineering Department
University of Chile
Santiago, Chile
mdiazq@ing.uchile.cl

***Abstract*—The success of the CubeSat nanosatellites space missions depends on all systems ability to perform properly in a harsh environment. A key component in every space mission is the flight software, which manages all the processes that must be performed by the satellite on its onboard computer. Literature shows that CubeSat missions suffer high infant mortality and many spacecraft failures are related to flight software errors, some of them resulting in a complete mission loss. Extensive software testing is the primary tool used by flight software developers, to ensure code quality and avoid such failures. Nevertheless, CubeSat developers tend to use COTS or flight-proven solutions which usually have low testing coverage. Nowadays, there is still some pending matter in the field of testing nanosatellites flight software and some of the most used solutions do not even report unit tests. To overcome the agile CubeSat development versus delivering quality software trade-off, we propose the use of fuzz testing techniques applied to the SUCHAI series of nanosatellites, being developed at the University of Chile. The successful application of this technique allowed us to find and solve many bugs not covered by classic strategies, such as unit testing and software in the loop simulation.**

***Index Terms*—Software Testing, Fuzz Testing, Testing, CubeSats, Nanosatellites, Flight Software**

I. Introduction

Initially, nanosatellites were conceived with a mainly educational purpose: the students can have the experience of developing and operating a satellite by themselves in the time frame of a college degree [1]. However, currently, the nanosatellites developing area has been expanding and opening to new scientific and technological challenges. Nanosatellites increasingly require more attention to their quality attributes to be successful in more complex missions. Specifically, the flight software of nanosatellites is a critical factor in determining a satellite's quality because it controls most of the tasks that must be executed while it is orbiting. If the software quality of a mission fails, it is probably that the whole mission fails too.

978-1-7281-5957-7/20/$31.00 2020 IEEE

In the space field, there are several testing techniques to verify flight software quality. However, the most advanced techniques are applied only to more complex systems, such as large satellites, rovers, or interplanetary missions [2]. In the state of the art, the most reported testing techniques applied to nanosatellites flight software testing are hardware in the loop simulation (HIL simulation) and software in the loop simulation (SIL simulation) [3], [4]. HIL simulation and SIL simulation methodologies can optimize the production process' overall costs in certain situations [5], [6]. However these techniques can be difficult to implement and execute, potentially dangerous to the hardware when executed in engineering or flight models, and time-consuming to set the environment up. Besides, the test cases must be predefined because these techniques are difficult to automate [5].

Fuzz testing is an automated software testing technique that consists of automatic random input generation to find software vulnerabilities [7]. In need of looking for an automatable and agile software testing technique applicable to nanosatellites, in this work we will study the usage of fuzz testing in the SUCHAI nanosatellite flight software [1].

II. Related Work

The most common testing techniques for CubeSats found in the literature are directly attached to hardware testing. Kiesbye *et al.* (2019) [3] present and evaluate an environment for HIL simulation and SIL simulation tests with the inclusion of the electrical domain for low-cost satellite development. The satellite tested was MOVE-II, developed at the Technical University of Munich. The obtained results are related to the verification of MOVE-IIs attitude determination and control algorithms, the verification of the power budget, and the training of the operator team with realistic simulated failures before launch. Additionally, they present how the simulation environment was used to analyze issues detected after launch and verify the performance of the developed new software to

address the in-flight anomalies before software deployment. The testing environment described in this work generates results for both hardware and software components of MOVE-II. According to the authors, the environment is potentially suitable for inclusion in a continuous deployment workflow where code changes trigger automatic tests on the hardware. However, they do not report full automation for test case generation.

Other software testing techniques found in the literature usually imply an exhaustive definition of test cases based on the requirements. Hishmeh *et al.* (2009) [8] show the design, implementation, and testing of the flight software for KySat-1, a picosatellite developed in the Kentucky Space consortium and launched in 2009. The testing methods applied to the software were strongly based on the requirements and documentation. Thanks to the testing methodologies applied to the flight software, most bugs were found in early stages of the development process. This begins with a requirement analysis. After this stage, the flight software team formulated a test strategy and began the testing planning. After the test cases generation, scripting, and execution, each bug found was reported. Although the software development team faced problems associated with time planning of students, it is not proposed a new strategy for the development or testing methodology itself, but a new organization strategy.

Johl *et al.* (2014) [9] present a reusable a command and data handling (C&DH) system as part of a series of CubeSat missions being built at the Austin Texas Spacecraft Laboratory (TSL), University of Texas. The key idea of this system is to support various system requirements, using a centralized architecture with one main flight computer controlling the actions and the state of the satellite. The flight software is its central component. To validate it, white box and black box testing techniques were planned and applied. The testing technique applied to the C&DH system was unit testing. Command execution testing and day-in-the-life testing were proposed to be applied as future work. Day in-the-life testing refers to verifying the functionality of the fully integrated satellite while a sequence of operations are being executed. We identify this type of testing as HIL simulation. They do not mention the methodology to generate the test cases nor an automated testing technique for the software verification.

Schoolcraft *et al.* (2016) [4] present a description and analysis of MarCO mission development. MarCO is a twin CubeSat mission developed by the NASA Jet Propulsion Laboratory (JPL) to accompany the InSight (Interior Exploration using Seismic Investigations, Geodesy and Heat Transport) Mars mission lander. MarCO refined the approach of all the development stages to solve the challenges of quickly building low-budget spacecraft to fly to Mars, relying on components reusability of previous missions. According to the authors, the MarCO flight software development occurred in a very tight loop focused on a hardware level since computer resources optimization was considered a development requirement. Therefore, the testing techniques applied to the flight software were mainly associated with HIL simulation.

The testing systems applied for the flight software of CubeSats are barely described in the literature of this area. The approaches found mention the use of unit testing, HIL simulation techniques or software tools that facilitate the data and command handling from the ground station, but they do not consider automated testing techniques that could be useful for time optimization, which is one of the most common problems for the flight software development. In this work, we propose and analyze the application of fuzz testing as an automated testing technique that follows the required agile development to execute space missions in the CubeSats field.

III. The SUCHAI Flight Software

The SUCHAI flight software architecture is based on the command design pattern adapted for implementation in the C programming language. Thus, the flight software acts as generic command executor and all functionalities are encapsulated as commands. The commands are requested by the client modules and derived to the invoker. The invoker enqueues the commands and take decisions about its execution to finally send the request to the receiver. The receiver executes the function associated with the command in the order they were enqueued [1]. The SUCHAI flight software architecture is illustrated by a UML communication diagram in Figure 1.

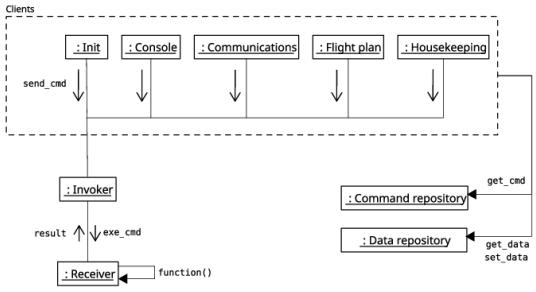

Fig. 1. The SUCHAI flight software architecture. Adapted from "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites," by C. Gonzalez, C. Rojas, A. Bergel, and M. Diaz, vol 7, pp. 126415, 2019.

Unit testing, integration testing, and HIL simulation are the primary testing techniques applied to the SUCHAI flight software development to improve and verify its quality. Unit testing was implemented using CUnit. The current unit testing system applied to the SUCHAI flight software is based on testing the main modules interfaces, but it contains at most four test functions for each module. The integration testing system of the SUCHAI flight software is based on some of the bugs that were found until present and consists of running the flight software with a specific configuration, sending the commands under test with fixed parameters, thus covering only particular use cases. In the case of HIL simulation testing, the software is being tested on the same onboard computer that will be installed on the satellite or the satellite flight model itself, which requires careful designing of the tests cases and setting up the environment (software, hardware, and facilities) previous to the execution of the tests in a controlled

environment. These tests have been included in a continuous integration system build using the GitLab CI/CD tools.

IV. Fuzz Testing

Fuzz testing is an automated software testing technique that consists of feeding a random input into a program to uncover system failures. Software failure is defined as an unexpected software behavior that gives a different result from the expected one. There are three main types of software failures: loss of service, incorrect service delivery, and system/data corruption [10].

A. Fuzz Testing on SUCHAI

As we explained in Section I, the SUCHAI flight software is considered a critical embedded system because it carries out the whole system control procedures of the SUCHAI nanosatellites. Therefore, we are interested in finding vulnerabilities associated with the system's availability and reliability. To study and analyze the robustness of the software, we present the application of fuzz testing on the SUCHAI flight software.

There are many ways to apply fuzz testing on the SUCHAI flight software, such as sending random input on functions, modules or commands. We chose to apply the fuzzing on commands because we can exploit the SUCHAI flight software architecture. As we explained above, the SUCHAI flight software architecture is based on the command design pattern, which means that all the functionalities are implemented and executed as commands. Thanks to its design, the software provides interfaces to receive commands as inputs through the satellite communication system (that can be emulated in the local loop), the serial console (or Linux terminal), the flight plan, and autonomously generated commands. This interfaces will be used to interact with a SUCHAI flight software running instance during the tests. On each test, we will analyze the result of sending a combination of random commands with a random number of parameters and/or random values of parameters. Thus, each test case should be composed by a sequence of commands.

We used the fuzzing architecture proposed in *The Fuzzing Book* (Zeller *et al.*, 2019) [11], which provides a Runner and Fuzzer classes. As described in Figure 2, the Runner executes some object with a given input. A Fuzzer feeds data into a consumer. The FSRunner class runs the SUCHAI flight software and sends commands to it. The data fed by the Fuzzer class is a sequence of random commands and their parameters. We are using the communication system interface to interact with the SUCHAI Flight Software. This interface uses the CubeSat Space Protocol (CSP) which provides a GNU/Linux and FreeRTOS implementation. The GNU/Linux implementation relies on the ZeroMQ library, so we commands can be sent using ZeroMQ sockets and the local loop.

B. Strategies

The implementation of the fuzz testing on the SUCHAI flight software is based on four strategies defined by the number of commands sent per sequence, the number of parameters sent per command, and the randomness to produce commands or parameters in a sequence:

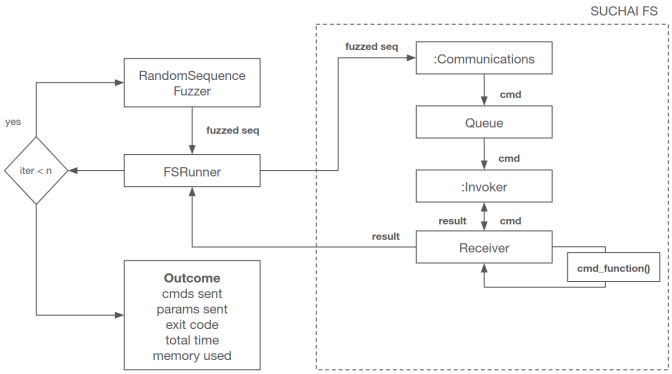

Fig. 2. Logic diagram of the fuzz testing implementation proposed and the communication system with the SUCHAI flight software. RandomSequence-Fuzzer generates the random sequence of commands to be sent to FsRunner. FsRunner initializes the SUCHAI flight software and sends it the commands to be executed. The SUCHAI flight software receives the commands through the communications module and executes them following its architecture logic.

Strategy 0: Random commands. Since the SUCHAI flight software provides a checking system for wrong names of commands, the key idea of this strategy is to prove the robustness of the SUCHAI flight software with random and possibly unknown commands. This can be achieved by providing sequences of random commands names without parameters. Thus, the implemented Fuzzer creates N random command names. These random names are stored in a list, which is sent to the implemented FSRunner class.

This strategy should not make the software to crash because there is command name checking system. Before the communications module sends the command object to the invoker, the program checks if the command exists in the command repository, iterating over the list of all the registered commands. If there is no matching name found for the command sent by the FSRunner, the command is not sent to the invoker for its execution.

Strategy 1: Random number of parameters. By providing sequences of known commands with a random number of parameters (including zero), this strategy searches for possible implementation errors in commands that are not considering the number of the passed parameters. Each parameter is a random value of a fixed type. The types are defined in the command implementation. These are int, long, unsigned int, float, and string.

In this case and the following ones, the Fuzzer receives a list of available commands in the SUCHAI flight software and the number of commands per sequence. Commands are chosen randomly from the list of available commands.

Strategy 2: Random values of parameters with randomly chosen types of values. Provides known commands with the exact number of parameters but random values. The types of the values are chosen randomly too, therefore they may not necessarily correspond with the expected types of values. The

goal is to find, mainly, errors on the implementations that may cause a crash because they do not check the values or the range of the variables in cases that it is required.

Strategy 3: Random values of parameters with defined types of values. With this strategy, we were looking for errors on implementations that have unchecked values characteristics, such as length, when it is needed. To achieve that we provided known commands with the exact number of parameters that each command receives, where each parameter is a fixed value of a defined type. Unlike the previous strategy, in this case the types of the values must correspond with the expected types.

V. Current Results

The preliminary results obtained from the execution of the strategy 0 and 1 mentioned in Section IV-B are analyzed in terms of the exit code for every sequence. We define a test case as the execution of a series of sequences with a fixed number of commands per sequence. Then, for each strategy, we executed 10, 100, 500, and 1000 sequences. For each test case, we executed 5, 10, 50, and 100 commands per sequence.

For strategy 0, the results show that the percentage of failure of sending random names of commands without parameters is 0%. Then, the results agree with the hypothesis that says the software is validating the command names before the execution.

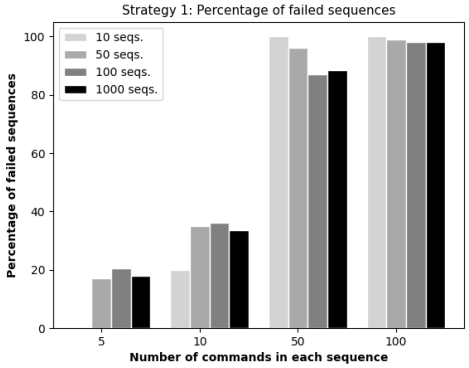

Fig. 3. Percentage of failed sequences of commands given a fixed number of commands per sequence for strategy 1.

The percentages of failed sequences on each test case for strategy 1 is shown as bar charts in Figure 3. The variable in the x-axis is the number of commands per sequence. The variable in the y-axis is the percentage of failed sequences for each test case. The bar charts are grouped by the number of sequences executed on a particular test case. For almost all cases, except one, as the number of commands increases, the percentage of failed sequences also increases. Since the random generation of commands and parameters uses a uniform distribution, the probability of choosing a command that could make the SUCHAI flight software to crash increases as the number of commands per sequence increases, which would explain the percentage raise. Increasing the number of test executions, that is, the number of tested sequences, we obtained less than 100% of failure, but still more than 90%.

VI. Conclusions and Future Work

In this work we explored the usage of fuzz testing techniques in the flight software of the SUCHAI series of nanosatellites by running a set of strategies. We were able to easily interact with the flight software and the fuzzer thanks to the clear and well documented software architecture. The test results showed that a large number of sequences failed during the tests execution which is a sign of active software bugs not found with previous testing techniques (unit-testing, integration test and HIL simulation).

The next steps in this research include the identification of the active bugs and the integration of the fuzz testing strategies in the SUCHAI CI/CD system to provide agility and automation. More advances strategies will be studied focusing in the intelligent identification of failure paths.
The proposed fuzz testing strategies and implementation may help current and future small and nanosatellite mission to improve their quality and thus, reducing mission risk.

Acknowledgments

We thanks Lam Research and the ANID FONDECYT Regular 1200067 for partially sponsoring the work presented in this paper. This work has been partially supported by the grants Fondecyt 1151476, Anillo ACT1405, and CONICYT-PCHA/Doctorado Nacional/2016-21161016.

References

[1] C. E. Gonzalez, C. J. Rojas, A. Bergel, and M. A. Diaz, "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites," *IEEE Access*, vol. 7, pp. 126 409–126 429, 2019.

[2] J. Finnigan, "A scripting framework for automated flight sw testing: Van allen probes lessons learned," in *2014 IEEE Aerospace Conference*, 2014, pp. 1–10.

[3] J. Kiesbye, D. Messmann, M. Preisinger, G. Reina, D. Nagy, F. Schummer, M. Mostad, T. Kale, and M. Langer, "Hardware-in-the-loop and software-in-the-loop testing of the move-ii cubesat," *Aerospace*, vol. 6, no. 12, p. 130, 2019.

[4] J. Schoolcraft, A. Klesh, and T. Werne, "Marco: interplanetary mission development on a cubesat scale," in *Space Operations: Contributions from the Global Community*. Springer, 2017, pp. 221–231.

[5] J. A. Ledin, "Hardware-in-the-loop simulation," *Embedded Systems Programming*, vol. 12, pp. 42–62, 1999.

[6] S. Jeong, Y. Kwak, and W. J. Lee, "Software-in-the-loop simulation for early-stage testing of autosar software component," in *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE, 2016, pp. 59–63.

[7] P. Godefroid, "Fuzzing: hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, Jan. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3363824

[8] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the kysat cubesat bus," in *2009 IEEE Aerospace conference*. IEEE, 2009, pp. 1–15.

[9] S. Johl, E. G. Lightsey, S. M. Horton, and G. R. Anandayuvaraj, "A reusable command and data handling system for university cubesat missions," in *2014 IEEE Aerospace Conference*. IEEE, 2014, pp. 1–13.

[10] I. Sommerville, *Software engineering 9th Edition*, 2011, vol. 137035152.

[11] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," in *The Fuzzing Book*. Saarland University, 2019, retrieved 2019-09-09 16:42:54+02:00. [Online]. Available: https://www.fuzzingbook.org