Automated functional system testing of Suomi 100 satellite

Juha-Matti Lukkari

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 16.1.2015

Thesis supervisor:

Prof. Esa Kallio

Thesis advisors:

D.Sc. (Tech.) Antti Kestilä

D.Sc. (Tech.) Juha Itkonen

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Juha-Matti Lukkari

Title: Automated functional system testing of Suomi 100 satellite

Date: 16.1.2015 Language: English Number of pages: 8+76

Department of Radio Science and Technology

Professorship: Space Science and Technology

Supervisor: Prof. Esa Kallio

Advisors: D.Sc. (Tech.) Antti Kestilä, D.Sc. (Tech.) Juha Itkonen

Large portion of launched Cubesats have failed early on their missions. Complete lack or inadequate system level functional testing of the satellites has been thought of being one large contributor to these failures according to some research done on the statistical data available about Cubesats. This thesis investigates potential solutions to mitigating these issues by the use of free open-source automated test framework. Presenting how we can do system level testing of Cubesats and what kind of tests could be done.
Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained. Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained.

Keywords: For keywords choose concepts that are central to your thesis

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Juha-Matti Lukkari

Työn nimi: Automatisoitu Suomi 100 satelliitin funktionaalinen systeemitestaus

Päivämäärä: 16.1.2015 Kieli: Englanti Sivumäärä: 8+76

Radiotieteen ja -tekniikan laitos

Professuuri: Avaruus tiede ja tekniikka

Työn valvoja: Prof. Esa Kallio

Työn ohjaaja: Prof Esa Kallio

Tiivistelmässä on lyhyt selvitys (noin 100 sanaa) kirjoituksen tärkeimmästä sisällöstä: mitä ja miten on tutkittu, sekä mitä tuloksia on saatu. Tiivistelmässä on lyhyt selvitys (noin 100 sanaa) kirjoituksen tärkeimmästä sisällöstä: mitä ja miten on tutkittu, sekä mitä tuloksia on saatu.
Tiivistelmässä on lyhyt selvitys (noin 100 sanaa) kirjoituksen tärkeimmästä sisällöstä: mitä ja miten on tutkittu, sekä mitä tuloksia on saatu. Tiivistelmässä on lyhyt selvitys (noin 100 sanaa) kirjoituksen tärkeimmästä sisällöstä: mitä ja miten on tutkittu, sekä mitä tuloksia on saatu. Tiivistelmässä on lyhyt selvitys (noin 100 sanaa) kirjoituksen tärkeimmästä sisällöstä: mitä ja miten on tutkittu, sekä mitä tuloksia on saatu.

Avainsanat: Vastus, Resistanssi,
Lämpötila

AALTO-UNIVERSITETET
HÖGSKOLAN FÖR ELEKTROTEKNIK

SAMMANDRAG AV
DIPLOMARBETET

Författare: Juha-Matti Lukkari

Titel: Arbetets titel

Datum: 16.1.2015 Språk: Engelska Sidantal: 8+76

Institutionen för radiovetenskap och -teknik

Professur: Kretsteori

Övervakare: Prof. Esa Kallio

Handledare: TkD Olli Ohjaaja

Sammandrag på svenska. Try to keep the abstract short, approximately 100 words should be enough. Abstract explains your research topic, the methods you have used, and the results you obtained.

Nyckelord: Nyckelord på svenska,
Temperatur

Preface

I want to thank Professor Esa Kallio and my instructors Antti Kestilä and Juha Itkonen for their good guidance.

Otaniemi, 16.1.2015

Eddie E. A. Engineer

Contents

Abbreviations

GPS	Global Positioning System
COTS	Commercial-off-the-self
Cal Poly	California Pylotechnic State University
1U	1 Unit CubeSat
P-POD	Poly Picosatellite Orbital Deployer
PCB	Printed Circuit Board
EPS	Electric Power System
OBC	On-Board Computer
COMM	Communication System
ADCS	Attitude Determination and Control System
DOA	Dead On Arrival
CFD	CubeSat Failure Database
NASA	National Aeronautics and Space Administration
SUT	System Under Test
ESA	European Space Agency
TLYF	Test Like You Fly
CONOPS	Operation Concept Document
RF	Radiofrequency
UHF	Ulta-High Frequency
AM	Amplitude Modulated
MCU	Microcontroller Unit
RISC	Reduced Instruction Set Computer
I2C	Inter-Integrated Circuit
CAN	Controller Area Network
GPIO	General Purpose Input-Output
SDRAM	Synchronous Dynamic Random Access Memory
PA	Power Amplifier
LNA	Low-noise Amplifier
CMOS	Complementary Metal Oxide Semiconductor
IC	Integrated Circuit
FM	Frequency Modulated
CSP	CubeSat Space Protocol
HK	Housekeeping
GOSH	GomSpace Shell
FTP	File Transfer Protocol
RTOS	Real-time Operating System
API	Application Programming Interface
Stdin	Standard Input Stream
Stdout	Standard Output Stream
SDR	Software Defined Radio
FFT	Fast Fourier Transform
CI/CD	Continuous Integration/Continuous Development

1 Introduction

1.1 Increasing interest in space

Since 1950s, mankind has ever increasingly started to set its foot into space [1]. Nations, industries, businesses, militaries, universities and even private entrepreneurs have sought out to benefit from the opportunities that space offers [1, 2, 3]. Great advancements have been made in technology and science to propel this endeavour even further [1]. Examples of such leaps in techology include sending first human into space in 1957, Moon landings in 1969, sending of probes into other planets in the Solar System like Mars and most recenlty getting the first images of Pluto in a flyby mission in 2013 [1, 4]. Use of space technology has also entered into household items through for example the use of the *Global Positioning System* (GPS) satellite network in mobile phones, cars, and so forth [5]. People and industries have began increasingly to be reliant on space borne technologies and devices [5].

Since the end of the 1990s new inventions have brought design and manufacturing of these technologies relatively closer to everyday people, away from the assembly sites of large nations and large organizations into laboratories run for example by university students. Advancements leading to this can be attributed to the space industry cathcing up with the advancements of electronics as well as cheap launch opportunities coming available. More concretely, development of the *CubeSat* nanosatellite concept in 1999 in CalTech has in recent years brought about hundreds of new satellite developers and hundreds of new space missions based on this nanosatellite concept. [3, 17]

These satellites typically are relatively small and usually use commercial off-the-self (COTS) components, yet are still capable to operate in space around Earth. Cubesats have in recent years emerged as a new viable platform for carrying out space missions. Due to their small size the launch costs are smaller and their use of COTS components makes them relatively fast and relatively cheap to design and manufacture. Several companies have taken interest into the concept and other organizations (like US military) have shown interest into them as well [7]. In addition, Finnish government recently passed a new law regarding space and is pursuing to create a new industry of space technology related companies in Finland [6]. The first Cubesats built in Finland in Aalto University have even created new space companies which are building their own satellites to be launched into space, like *Iceye* for example [8]. Nonetheless, usually most of these satellites have been produced by Universities and over 400 Cubesats have already been launched in total since 2000 [17]. The trend seems to be so that more and more CubeSat missions are to come and they are starting to take a clear share of the space industry market [18]. Already in 2014, approximately half of the flown space missions in that year were CubeSats [17].

1.2 Substantial proportion of failures in CubeSat missions

Even though the CubeSat concept has rapidly created large amount of new space missions, large portion of those launched missions have ended in failure due to various reasons. Several surveys done in recent years have found a failure rate of 40 % for new CubeSat missions. Yet, if the satellites were manufactured by teams with earlier experience in space missions, the failure rates were considerably lower. Suggestions have been made in these surveys that the missions that have failed haven't done proper functional testing on ground on a system level or that such testing has been missing completely. As an example, research done by Swartwout in 2013 into statistical data of first 100 flown Cubesats discovered that over 40 % of Cubesat missions ended in failure. This research suggests that majority of these failures could be attributed to inadequate testing of the satellite in a flight-equivalent state on ground. It was believed that functional testing of the whole integrated satellite system has been lacking completely or done in a very limited sense. This allegation could been made at least for those missions where the satellite was never contacted from the ground station after the launch, as many of the failures in these missions were attributed to failures in system integration. Such as the solar panels not being properly connected, insufficient power generated for the transmitter, unrecoverable processor errors and so forth. [3, 17, 19, 20]

Some examples of early life failures include Aalto-2 satellite, the first Finnish satellite flown in space [source]. Though the concept of Cubesats shows promise, there are still some challenges for the concept to become a truly reliable alternative approach for scientific and commercial missions alike as opposed to the more traditional time and resources consuming approaches in space mission development [3].

1.3 Suomi100 Cubesat

The satellite involved in our research is called *Suomi100* which is a one unit Cubesat. About the mission plaaplaa. Finland centennary. Images of Finland from Space. An artistic presentation can be seen in Figure 1 below.

1.4 Research purpose and goals

The aim of this thesis is to investigate *how to carry out functional system testing in order to improve CubeSat reliability.* Further, we wish to automate the testing and to do it systematically, just as the verification tests for example for mechanical stress are done systematically in an automated fashion. So too it would be preferable that the functional system tests could be done automatically in a systematic manner.

On the technology point of view, the goal will be obtained by developing new generic Python library and test suites which can be used in *Robot Framework* [42] along with appropriate test setups.

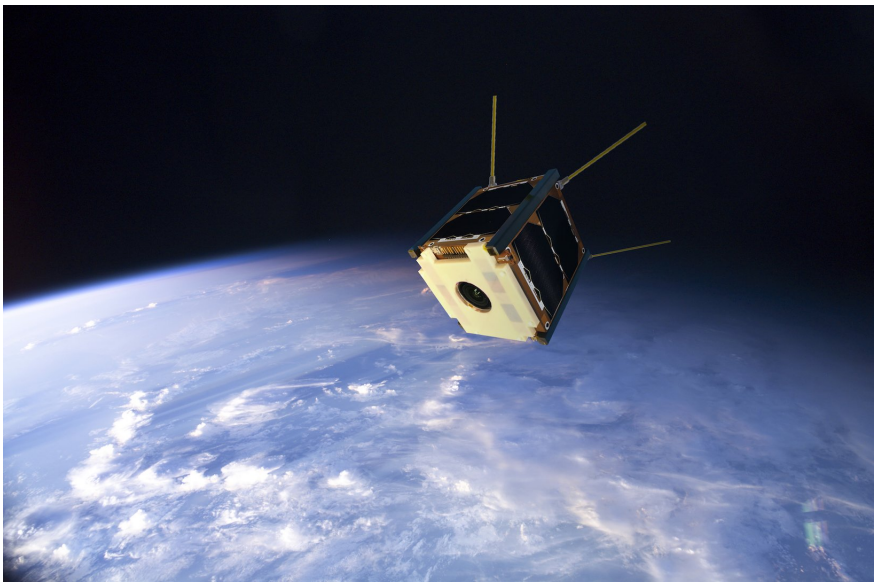

Figure 1: Depiction of Suomi100 satellite in space. Courtesy of Jari Mäkinen. [46]

1.5 Main questions and problems

The main problem is the unreliability of Cubesats, which the thesis tries to partly solve from the standpoint of system integration testing. Could this type of testing detect unrecoverable failures in the satellite operation and system integration and could we in addition verify that the satellite meets out its functional requirements?

1.6 Outlining the scope of research

We wish to study the use of one industry-proven automated acceptance framework to carry out the automated functional testing on Suomi100 and to investigate the use of industry-proven testing philosophies and methodologies into functional system testing with the satellite.

In conjunction with the space industry test methodologies, we attempt to some degree simulate the environment related to functional operations of the satellite. In addition, the simulation has to take into account the relatively small funds of an university led project.

2 Background

2.1 Cubesat failures

The CubeSat project was started in 1999 in California Polytechnic State University (Cal Poly) [7]. Over 100 universities and other organisations have since contributed to the project. The purpose of the project is to provide a standard for design of picosatellites in order to decrease development costs and to make accessability to space easier [7]. In fact, number of launched CubeSats has increased quite dramatically over the past few years [17] and it has been estimated that the number of CubeSat missions will increase in the coming years [18]. Yet, large portion of these missions have failed due to various reasons [3, 17, 19]. With only an average of 20 % of missions being able to complete the full mission envisioned for them [17].

2.1.1 CubeSat satellite

The CubeSat project defines a CubeSat to be a picosatellite with dimensions of $10 * 10 * 10cm^3$ and with a mass up to 1.33 kg [7]. Satellite of this size is considered to be *one unit* (1U) CubeSat. These units can be stacked together to form larger CubeSats, with some satellites consisting even of 12 units. Three units seems to statistically be the preferred size for a CubeSat [17].

Another important part of the concept is the *Poly Picosatellite Orbital Deployer* (P-POD), which is a Cal Poly's standardized CubeSat deployment system [7]. This deployment system is integrated to the launch vehicle and the springs in the system release the picosatellites into space [7]. Usually a launch vehicle carries some primary payload, which is much larger than the CubeSats [9]. If an extra weight can be launched along the main payload, then the deployment pods with the CubeSats are integrated into the launch vehicle as secondary payloads [9].

Though these picosatellites are considerably smaller than most of the "traditional" satellites, they nonetheless are able to perform the regular operations of a satellite [11]. As the classification, *picosatellite* defines, CubeSats are satellites in miniature size. Even though smaller, the same general class of subsystems are part of CubeSats as they are part of larger satellites [11]. Subsystems containing electronics usually follow *PC/104* standard which defines form factor and computer bus [12]. A single subsystem in a CubeSat can fit into one *Printed Circuit Board* (PCB) following the PC/104 standard [12].

Figure 2 illustrates the internal structure of Suomi100 CubeSat and the different subsystems that are integrated into the satellite.

From this figure we can identify five subsystems that are common to all satellites. Power to the satellite is produced by the *Electric Power System* (EPS) and this subsystem consists of the solar panels and batteries in the satellite. The subsystem usually in addition has some power regulation and power distribution features along some features for reliability. [10]

The central computer of the satellite is called *On-Board Computer* (OBC) and the task of this subsystem is to orchestrate the operation of all the other subsystems.

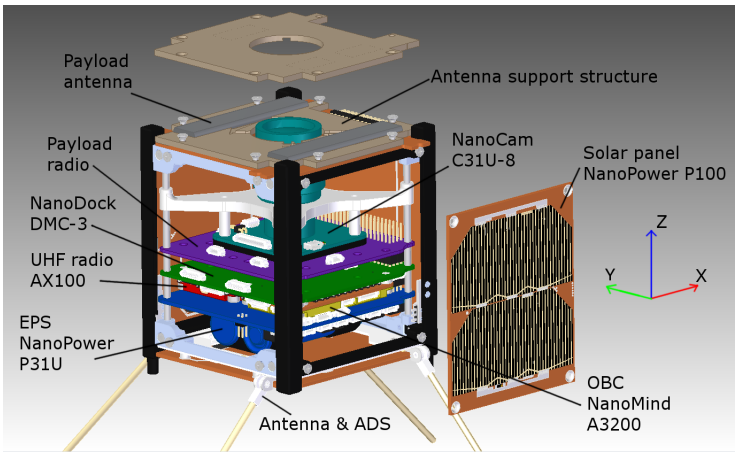

Figure 2: Depiction of Suomi100 satellite subsystems. Courtesy of Aalto University. [46]

In addition, the processing of commands received from the ground station and the routing of them to the appropriate subsystem is the task of the OBC. [10]

The *Communication system* (COMM) is responsible for communication with the ground station. This subsystem usually has a computer of its own for processing the received radio signals. The antennas are also part of this system [10].

The proper orientation of the satellite is controlled by *Attitude Determination and Control System* (ADCS). For CubeSats, the orientation can be controlled either by mechanical reaction wheels, magnetorquers or by some other methods. The purpose of this system is to keep track of the orientation of the satellite and to change it according to commands received from ground station. [10]

In addition to these subsystems, *Support Structure* forms the subsystem which integrates all of the subsystems into a single mechanical structure [10]. The CubeSat standard defines the dimensions and materials for the support structure [7].

2.1.2 Failure rates of CubeSats

Over 400 Cubesats have flown as of 2017 [17]. Few studies in recent years have been carried out to investigate the statistics of flown Cubesat missions. These studies looked for the percentage of failed missions and which subsystems contributed to each failure. Out of these failures, the amount of *dead on arrival* (DOA) missions where the satellite was never contacted from space were also identified. Most active contributor to this topic has been Michael Swartwout and the representation of statistics of CubeSat failures in this thesis is mainly based on his work [3, 17, 19], as not too many papers have yet been published regarding this issue.

A study titled *The First One hundred CubeSats: A Statistical Look* and published in 2013 was one of the first papers to analyze the statistical data of flown CubeSats. Out of the first 100 CubeSat missions investigated between years 2000-2012, a total of 34 had failed. Out of these failures, third were never contacted after they were released into space (DOA cases). Figure 3 shows the clear amount of failed missions

out of the first 100 flown CubeSats. [3]

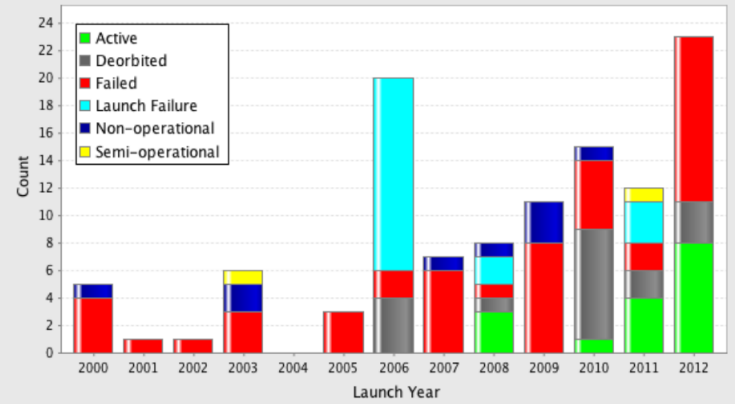

Figure 3: Statistics of first 100 flown CubeSats. [3]

Same author has been continuing yearly to publish papers about the statistics of CubeSat failures, as new missions are flown each year [17, 19]. In these papers Swartwout has included statistics about other secondary payloads as well, though majority of secondary payloads have been CubeSats. A study published in 2016 titled *Secondary Spacecraft in 2016: Why Some Succeed (And Too Many Do Not)* identified out of all CubeSats flown between 2000-2015 that reached orbit 21 % DOA cases and 9.8 % cases where the spacecraft was lost early in its life, meaning that communication with the satellite was established but no primary operations could have been executed. When breaking down the statistics to categories based on the type of satellite and mission developer (new University teams, traditional contractors, experienced University and government teams, constellations), it was found that for the new University teams flying their first satellite, the failure rates were as follows: 44.1 % DOA, 16.2 % early loss and 16.2 % mission success. On the other hand, for the CubeSats built by traditional contractors with established practices for integration and testing the numbers were: 6.3 % DOA and 6.3 % early loss. On the other hand, when the new University teams educated by their first failure flew their second satellite, the rates for DOA failures halved. [17, 19]

Below Figure 4 shows the statistics of failures for CubeSats between 2000-2015 flown by new University teams sending their first satellite, excluding those missions where the satellite was lost due to launch failure.

For contrast, same statistics for CubeSats built by traditional contractors is shown in figure 5 below.

2.1.3 Contribution of different subsystems to satellite failures

Some study has been carried out by Swartwout in one of the aforementioned papers to investigate the contribution of different subsystems in CubeSat failures. In the paper *The First One hundred CubeSats: A Statistical Look*, the subsystems that were thought of been the cause of the failure were identified as follows: a configuration or interface failure between communications hardware (27%), the power subsystem

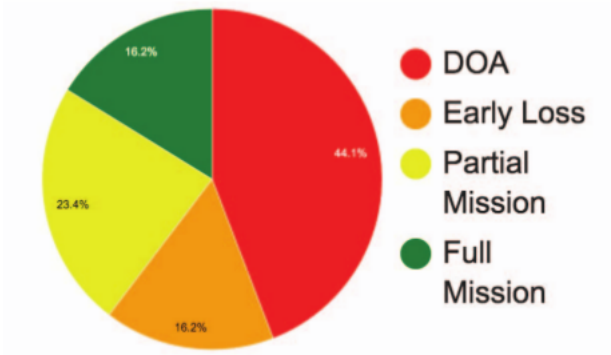

Figure 4: Statistics for CubeSats flown between 2000-2015 that were constructed by University teams without prior experience for satellite construction. [17]

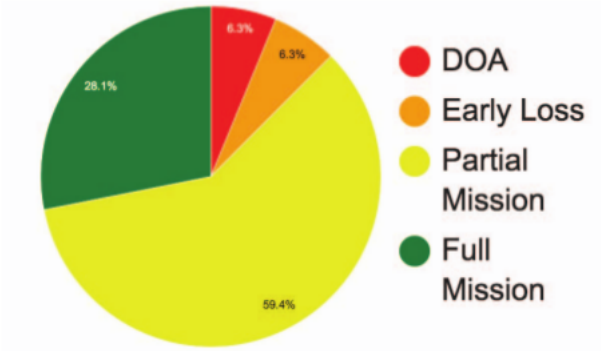

Figure 5: Statistics for CubeSats flown between 2000-2015 that were constructed by traditional contractors with extensive experience of satellite construction. [17]

(14%) and the flight processor (6%), or COMM, EPS and OBC subsystem. A failure in a subsystem means in this context that the whole satellite is lost due to the failure. A failure in the OBC can mean for example that the processor doesn't reboot anymore or gets unrecoverable stuck in some way. For EPS the error can mean for example that power is not being transferred to the satellite from the solar panels and for the failure in COMM subsystem it can mean for example that there is insufficient power for the antennas to close the link with the ground station. [3]

Based on the satellite developers believes about causes of failures, another study was carried out by Langer et al. in 2014 [20] to investigate in more detail the contribution of different subsystems in CubeSat failures. This study by Langer also used statistical data of CubeSat failures obtained from CubeSat Failure Database (CFD), at that point comprising data about 178 CubeSat missions. With this data a reliability estimate for different subsystems was calculated using Kaplan-Meier estimator for nonparametric and parametric analysis. In addition, a parametric model for total CubeSat satellite reliability was devised. Figure 6 below depicts the subsystem contributions to satellite failures for the 178 CubeSat missions. Three main subsystems causing failures were identified to be in order: EPS, OBC and

communication systems, in accordance with Swartwout research, but with different percentages as EPS being the main contributor to failures [3]. [20]

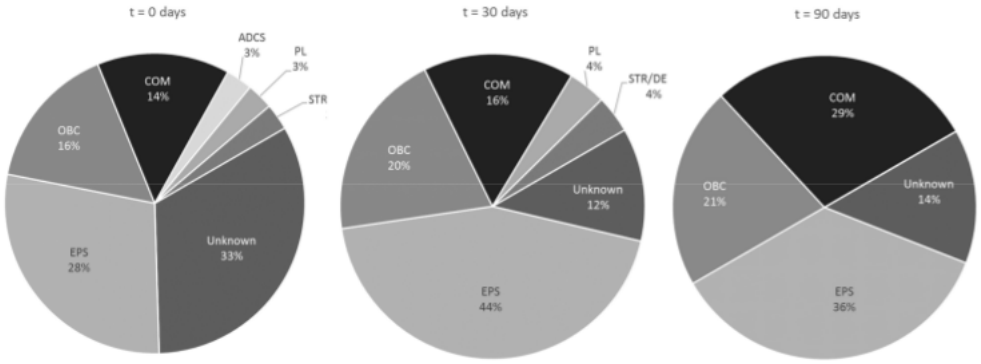

Figure 6: Developers believes on the contribution of different subsystems to satellite failure. From left to right the charts present failure contribution data for 30, 60 and 90 after launch. [20]

The statistical data gathered from questionnaires sent to 987 satellite developers (with 113 returned fully completed) showed that there was a belief that within the first six months there was a 50 % change that the satellite fails. Yet, the developer's belief seems to be too optimistic when compared to the data gathered from CFD. Nonetheless, from largest to smallest probability for causing system failure the main subsystems were identified in the order COMM, EPS and OBC.[20]

2.1.4 Needs for the system level functional testing

The aforementioned studies made some anecdotal guesses to what could have contributed to the failures in the satellites so that the missions failed either partially or completely. The current data doesn't clearly prove these guesses, it has been believed by Swartwout and others that system level functional testing of the satellites has been lacking completely or has been inadequate [3, 17, 19, 20].

Based on his study in 2013 [3], Swartwout came to a strong belief that the critical failures in the subsystems were caused by poor functional integration. Notably, out of first 30 identified DOA cases, 24 were CubeSats made by university teams. In addition, based on his discussions with project managers and faculty leaders it was noted that university teams constructing CubeSats have the misconception that the satellite works as expected the first time it is assemled together and thus no system level functional testing is performed. In this paper it was believed that operational tests demonstrating "a day in the life of the satellite" would be just as necessary as the vibrational tests to certify a CubeSat ready for launch. In addition, testing of recovery from resets, power management, startup sequences etc. would be important operations of the satellite to test. [3]

In later papers Swartwout has been less reluctant to make these claims directly, yet still identifying the large number of failed CubeSat missions coming from university-led satellite teams [17, 19]. As an example, the ORS-3 mission flown in 2013 consisted

of 28 secondary payloads and out of these 13 were done by new university teams flying their first satellite and 15 were done by traditional contractors [17]. While almost all (11 out of 13) of the university constructed CubeSats failed, only one CubeSat built by a traditional contractor failed. Furthermore, all of these satellites had to go through the same vibrational and thermal tests and in addition were subject to mission readiness reviews by the *National Aeronautics and Space Administration* (NASA) and/or Department of Defense of United States. Thus, some practices applied by the experienced contractors in satellite development were quite possibly missing from the university built CubeSats.

In addition, David Voss of the Air Force Research Lab speaking more recently in the 31st Annual Conference on Small Satellites held at 7th of August 2017 about CubeSat reliability said that based on his experience with student and other small satellite projects, a core set of tests for power, communication and other subsystems would be needed [21]. Furthermore, Michael Johnson, also a participator in the aforementioned conference and chief technologist at NASA's Goddard Space Flight Center has been since 2017 working at NASA on making a reliability iniative to determine the best ways to improve CubeSat reliability [21, 22]. He noted though that the goal is not apply the same rigorous assurance procedures used earlier in larger and more expensive spacecrafts, but to design new procedures and keep some of the familiar methods that can be useful.

2.1.5 Comparison of CubeSat failures to failures with larger spacecrafts

Besides Cubesats, failures have happened to the more traditional spacecrafts as well. In fact, the history of space industry as a whole is filled with examples of failed missions [source]. In recent years, for example several Mars landers have failed during landing phases of the mission [23]. As an example, Mars lander *Schiaparelli* crashed on the Martian surface in 2016 when a sensor used to measure distance to the ground read a negative value and shut off its descent thrusters [24].

Earlier research about the on-orbit failures was carried out in 2005 [25]. It investigated failures of 129 different spacecrafts between the years 1980 and 2005, before any flown Cubesats, and it noted that though these spacecrafts had gone through intensive testing and used the most recent technologies available, there where still cases where the spacecraft failed early during its mission. The investigation also found that adequate testing on ground could mitigate these failures as it was noted that the early failures could have been caused by inadequate testing and inadequate modeling of the environment where the spacecraft operates in. These conclusions in fact seem to be similar to what some of the surveys done on CubeSat failures indicate. [25]

A research conducted in 2008 analyzed the contributions of different subsystems in failures of 1584 satellites that were launched between 1990 and 2008 [26]. Solar array deployment and failure in the communication system were the major contributors to satellite failures for satellites that failed before 30 after launch. After much longer operation time, the main subsystems contributing to failures were identified to be the ADCS and COMM. Some similarity to the subsystem failures with CubeSats

can be drawn from here, with the communication subsystem and the solar panels playing a crucial role in infant mortality of CubeSats.

Further prove for the need of extensive testing was found after NASA initiated in the 1990s a more streamlined verification strategy based on the best commercial practices, commonly known as *"Faster, Better, Cheaper"* [27]. This led to poor results with commercial satellites that were launched during that period and a return to the more rigorous specifications and standards was expressed [27]. In addition, a study conducted during this period in 1999 called *"When Standards and Best Practices Are Ignored"*, found that out of 50 major space system failures 32 % were related to inadequate verification and test processes [28].

In conclusion, based on the experiences found by traditional space industry, the allegation that many CubeSat failures are due to poor system integration level testing could be correct.

2.2 General testing practices

Testing of hardware ? Embedded system testing is software testing? Thus our focus in this thesis is on software testing. Since 1970s, testing has been considered as a specific branch of software testing [14, 30]. More advanced techniques and methods for testing have been developed over the decades since [14].

NASA has contributed to the practices used in software testing through the its effort in the Apollo program for example [30].

First a relevant question, *who do we need to perform testing?* One reason is that humans make errors and often make optimistical assumptions about their work. Another aspect would be to call testing as a method of proving that the system under test works as we want it to work. Just as a scientist carries out experiments to prove his theory, so too testing is done to prove that the system works as expected. [15]

Another important aspect of testing is to find defects in the system and to recognize where they exist so that they can be corrected effectively [13, 14]. A book by Glenford J. Myers titled *The Art of Software Testing*, defines software testing as *"Testing is the process of executing a program with the intent of finding errors"*, which is a general enough definition to contain many aspects of software testing [14].

2.2.1 Methods of testing

Various different methods for software testing exist. So called box approach is one common method for testing. Testing can either be done automatically by some computer run script or manually by a tester who follows a specified test plan. [13] Below are some of the testing methods explained in more detail.

Black box testing is done with no knowledge about the internal structure of the *System Under Test* (SUT), which can be a single function of a software or the whole integrated system. A select set of inputs are given to the system and from outputs we see how did the system perform. If the outputs were what we expected, the system passed the test. Black box testing is usually implemented when we are interested in the functionality of our system under test and test cases are designed

from the specifications and requirements of the SUT. There exists some philosophy in choosing the right inputs to the system or software. With *exhaustive testing* all possible input combinations are investigated, which usually leads to combinatorial explosion and the testing all of them can for example even take millions of years! *Boundary-value testing* solves this issue by having some logical set of combinations and not all possible ones. Figure 7 below illustrates black box testing method. [13]

White box testing is done with interest about the internal structure of the SUT. Testing of the internal functions rather than the expected functionality is the goal of white box testing and test cases are derived from the code of the SUT. Usually this type of testing is performed at the smaller component or unit level of the system. Figure 7 below gives an illustration about white box testing. [13]

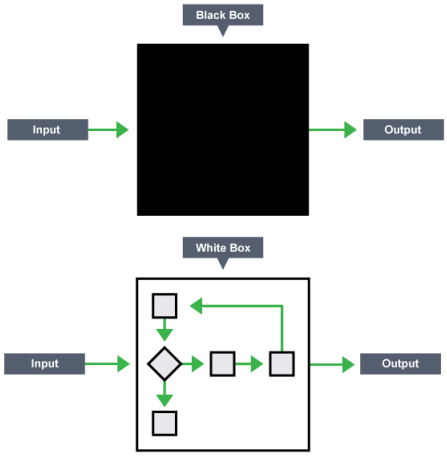

Figure 7: Black and white box testing methods. [source]

2.2.2 Levels of testing

Testing can be carried out at different levels of the system and at each level we investigate different aspects of the system. Usually testing of an entire software is done by starting from smaller parts and gradually going into larger components of the system. [13, 15] One can define different levels of testing as follows [13, 15]:

Unit testing

Unit testing is the most basic level of testing. On this level, individual components of a software or system are tested separately. For example, testing the outputs of a given software function is considered a unit test. Usually these tests are written or performed by the person who also wrote that particular part of the software. Black box and white box methods are usually applied for unit tests. [13]

Integration testing

Integration testing tests the functionality of larger software or system components, consisting usually of several smaller units. With this level of testing we ensure that the smaller units interact with each other properly and that the bigger component

itself works properly. Both black box and white box method can be applied to carry out the tests. [13]

System testing

On this level, testing is done on a complete integrated system to see whether it conforms to the requirements specified for it or not. With this testing we see whether the integrated parts of the system work together and also see how the whole system functions. Black box testing is usually applied at this level. [13]

Acceptance testing

Acceptance testing is usually performed after the system in question has passed system testing. At this point testing is usually performed by some outside team that wasn't involved in the development of the system or software. This team could mean customers or there could be a separate testing team to ensure the functionality of the product. Blac box testing methods are applied at this level.

2.2.3 Types of testing

Besides the method and level of testing chosen, there exists multiple different types of testing that can be performed. The most common ones are the following [13]:

Functional testing

Functional testing is performed when we are interested in knowing what the system under test does based on our input to it. Black box testing methods are mostly applied here. Functional testing can be done at all levels of testing.

Non-functional testing

Non-functional testing on the other hand is interested in how the SUT operates, rather than what it actually does. Several different testing types can be considered to belong under this, like e.g. performance testing or security testing. Both black and white box methods can be applied to this type of testing.

Performance testing

Performance testing is done for the interest of knowning how stabile and responsive the system is under a certain load. This testing can be done at all levels and the methods can vary.

Regression testing

Regression testing is usually performed after the software has changed from the previous version which had been tested. For example, when a new feature is added or some defects are fixed, regression testing is done to see whether the old parts of the software still work as expected. Usually a fixed set of unchanging tests can be executed once every change has been made to the software. [13]

Smoke testing

Smoke testing is carried out to verify that the most important parts of the system work properly. Usually the test set is small as we are only interested in seeing if anything fundamental is not working in the system.

System integration testing

Rewrite. Use [16].

Wikipedia: SIT consists, initially, of the "process of assembling the constituent parts of a system in a logical, cost-effective way, comprehensively checking system

execution (all nominal & exceptional paths), and including a full functional check-out."[1] Following integration, system test is a process of "verifying that the system meets its requirements, and validating that the system performs in accordance with the customer or user expectations."

In this thesis, we have chosen *automated functional system testing* to cover the type of tests that we performed for Suomi100 satellite. *Automated* refers to the fact that we are using one automated testing framework for test execution. *Functional* comes from that we test what the satellite does based on our inputs and commands to it. *System level* refers to the fact that we are performing testing on the whole integrated satellite. Testing is carried out this way because, as mentioned earlier in section 2.1, this kind of testing has possibly been lacking in the previous CubeSat missions and carrying out these tests could possibly mitigate failures that occur in the early life of a satellite.

– How much testing in general is done and how much automated testing has been implemented in the industry sector?

– The value gained with automated testing and with testing on a system level. Acceptance testing seems to be common for example.

2.3 Testing of complex systems

2.3.1 Embdedded systems

Something general about testing of systems with hardware and software. Some examples on testing of embdedded devices in industry sector.

2.3.2 Space Industry

In the 1950s first man made devices were sent into space by the USSR and United States [source]. Since that time, space industry has seen a gradual growth with perhaps the biggest and most expensive endeavours being the manned missions to the Moon, the U.S. shuttle program and the International Space Station being funded by several nations [29]. During the evolution of the industry, manufacturers have had to develop and improve the practices in software and hardware design and testing as the devices have become more complex and reliability has become an issue [30]. For example, software engineering as a specific branch of computer science emerged during NASA's Apollo program [30].

In addition, space as an environment itself provides extra challenges for maintaining proper reliability of spacecrafts sent there. Variations in temperature as well as particles carried by the solar wind put unique demands for reliable design. These devices operating practically out of our physical reach impose further demands for system reliability. If in a satellite orbiting at an altitude of 500 km happens an unrecoverable processor error, there is no practical way for us to go there and physically press the reset switch to get the satellite operating again. [10]

During the course of the Apollo program, NASA adopted a four level software testing practice [30]. In NASA, on different levels of system development different environments and different teams are used for testing [31]. Selitykset eri tasoista [31] mukaan. At the lowest level, unit tests of the software are carried out by the software developers and on higher levels completely separate teams are used to carry out testing. Simulators and testbeds are used when testing assembled subsystems [31, 32]. For example, a system testbed was used to test operation of singular and several subsystems of the Cassini-Huygens space probe [32]. Several inputs to the subsystems simulated the space environment while tests were being carried out. On the highest level the whole spacecraft is assembled and is tested by testing the system with different scenarios of satellite operation [31, 33]. For example, downlink procedures, maneuvres, payload operations and so forth are tested on this level. Below is 8 describing the levels and methods of testing at different levels of spacecraft development.

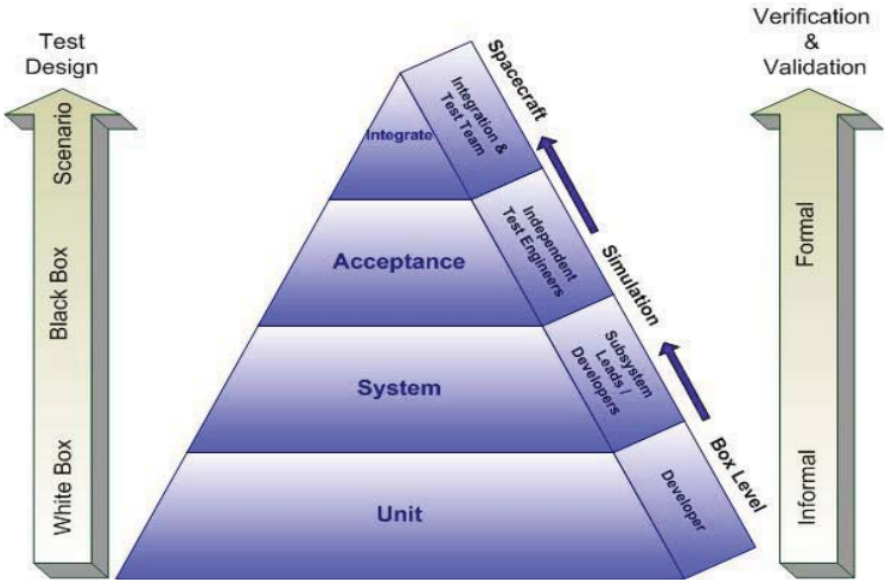

Figure 8: Illustration of spacecraft testing on different levels of system development. [31, 33]

At the *European Space Agency* (ESA) the preferred levels for testing of the spacecraft are equipment, subsystem, element, segment and overall system [34]. ESA also states that a system verification by testing shall consist of testing of system performance and functions under representative simulated environments [34]. As can be seen, testing is done in the same vein as what is done in NASA.

Emphasis on testing the highest level of assembly or in other words, the whole spacecraft has always been a NASA priority [35]. A mantra commonly used in space industry has been *"Test like you fly"* or TLYF, meaning for one that spacecrafts should be tested on ground in the same way as they would be operated in-orbit [27, 33]. On general, the TLYF philosophy provides a basis for acquiring and verifying a system and gives a mission-centric focus on space system validation and verification

[33]. As such, same software and same hardware should be used in testing as will be used when the spacecraft is launched to orbit [27]. One test on system integration level using this philosophy is commonly referred as *"day in the life"* or operational scenario testing [27, 33].

In "day in the life" testing, tests are derived from mission operations requirements documents, a document called *operations concept document* or CONOPS is commonly used for this type of documentation [33]. The focus with this testing is on verifying whether the space and ground segments can accomplish the mission as it was envisioned in these documents. The test involves having the integrated and assembled spacecraft on ground being flown in a flight-like manner to the extent feasible and controlling and communicating with the spacecraft from the ground station in the way that has been envisioned in the mission operations requirements document [33]. This type of testing has been deemed necessary as many failed space missions had been succesfully tested to meet all their requirements, but were not tested to verify succesful completion of mission objectives [33]. This test is required by NASA GSFC and ESA to be performed before a spacecraft can be verified for mission readiness [33, 34].

Explain what NASA does at each level, unit, subsystem, integration, system, acceptance & operational etc. ESA ECSS-E-ST-10-02C. NASA Systems Engineering steps and procedures (design reviews, SAR, SIR etc.).

2.4 Tools for automated functional system testing

2.4.1 Test automation frameworks

Software test automation has among software projects been a topic of interest for a past few decades [37]. It has been proclaimed as a solution for decreasing costs related to testing and enabling release of human resources for other tasks [38, 39]. Test automation can be performed on many levels of testing, from unit level to acceptance and beyond. It has been found to be most useful in automation of repetive tasks and in automating execution of repetive test cases [39]. Several softwares and frameworks for test automation have been developed over the brief history of test automation [37, 39].

A test automation framework is an integrated system that sets the rules of automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. When some changes are made to the system under test, only the test cases are needed to be modified. [40]

The common practice is that test cases are written into separate scripts with a scripting language specific to the framework [40, 42]. Function libraries are written into their own source code files with some of the more common programming languages like Python, Java, C++ and so forth [42, 41]. The test scripts then call for the functions in the function libraries to perform the actual automation [41]. For example, a script could call functions for sending commands to the system via network connection.

Several different test automation frameworks exist based on different types of

test case generation techniques. Frameworks based on *Keyword-driven* testing use keywords. Frameworks utilizing *Data-Driven* testing use something else. In *Model-Based* testing happens something [modelbasedtesting].

2.4.2 Robot Framework

Robot Framework is generic test automation framework for acceptance testing originally developed in Nokia Networks [42]. The framework emerged from a Master's thesis written by Pekka Klärck for one Finnish software testing consultancy company known as *Qentinel Oy*. Title of his thesis was "Data-Driven and Keyword-Driven Test Automation Frameworks" and it was written in 2005 [42, 43]. In turn, the writer of this thesis at hand has also been working at Qentinel and thus has become quite familiar with Robot Framework. This is one of the reasons why Robot Framework was chosen for test automation of Suomi100 satellite.

Robot Framework is in addition open-source under Apache 2.0 license and the modularity of the framework allows people and companies to write their own testing libraries either with Python or Java. The core of the framework is implemented with Python. The modularity and flexibility of the framework has made it possible to use the framework to do test automation on various different projects. Some companies like ABB, Nokia, Kone, Metso, Axon, Zilogic and others have used Robot Framework in testing of embedded systems. While other companies like Finnair have been using it to test their web based applications. Some companies like ABB, Metso and others are performing their testing with Robot Framework in many different areas. U.S. Naval research laboratory has also been using the framework with their SAGE multi-agent system. [42]

Based on how many companies have been confidently using the framework [42] and that it has been used in many different areas (embedded systems, web applications, etc.), we feel confident to develop test automation for Suomi100 satellite with Robot Framework. In addition, the framework being open-source makes it even more appealing for this task [42]. Future CubeSat projects could use Robot Framework as well and possibly with the generic libraries that were created in this thesis.

Robot Framework uses keyword-driven testing technique and the test scripts have a tabular data syntax. Below in Figure 9 a Robot Framework test suite script is shown.

In Figure 9, the number (1) on the script shows how function libraries are included to test execution. These libraries can simply be Python files or Python classes. In that case, a simple Python file can be included to test execution by defining a relative path and the filename. Python modules which are included in operating system *PATH* variable can be included directly.

Number (2) on the script illustrates how test suite setup and teardown scripts are included. These calls to start and close the test execution can as well be Python functions included in the function libraries. Call for *Suite Setup* defines what operations are peformed at the start of the test execution, like opening of the software under test, initiating network connection to the software and so forth [44]. Calling for *Suite Teardown* defines what actions are performed when test execution

```
*** Settings ***
Library         String
Library         AutomationLibrary
Suite Setup     Start Suite
Suite Teardown  Close Suite

*** Test Cases ***

Set satellite orientation
        Verify Connection
        Set Orbital Elements    10      20
        Set ADCS                1       2       3


Measure particles with payload
        Verify Connection
        Startup payload         10
        Measure particles       HIGH    100000
        Store Measurment        /flash/data/measurment.dat
        Downlink Data           /flash/data/measurement.dat
```


Figure 9: Example of a Robot Framework test suite with two test cases.

ends [44]. Like closing of the software which we are testing and so forth.

Number (3) shows in what way the test cases are defined. The names of the test cases are arbitrary. Though they should be named differently from each other and the naming should represent the activity of the test case [44].

Lines next to number (4) present how keywords are called. The keywords can be direct calls to Python functions or methods of the same name or they can be other keywords defined in some Robot Framework script file. The underscores and letter cases in the Python function definition are translated so that the representive Python function can be called with the keyword in any way the keyword is written [44]. Spaces and letter cases don't matter in the keyword when calling for a Python function from Robot Framework.

Under number (5) the parameters for the representive keywords are defined. Parameters are separated from the keywords by arbitrary number of tabular spaces as well as from each other [44].

2.5 Suomi100 satellite mission

Suomi100 satellite was conceived in 2015 in the interest of celebrating Finland's 100 years of independence [45]. The original design called for a 2U CubeSat, but was later changed to a 1U CubeSat. The mission demands to have two payloads on board the satellite. First payload is a white ligth camera in the interest of taking images of Finland from space and the second one is a science instrument which measures the radio static in the ionosphere [45, 46].

2.5.1 Mission requirements

The requirements of the mission are presented in this section. The first requirement defines the functional requirement of the mission as:
"Take images of Finland and measure RF radiation caused by northern lights."

Table 1 on the next page shows requirements derived from this requirement.

Table 1: Suomi100 functional mission requirements derived from the requirement to take images of Finland and measure *Radiofrequency* (RF) radiation caused by northern lights.

1st Derivation	2nd Derivation	3rd Derivation
Take one image of Finland per day	Capable of pointing camera towards Finland	Camera points with 5 degree accuracy
	Must compress images for faster downlinking	RAW, BMP and JPEG output formats
	Image resolution shall be adequate to discern geographical features	60 m / pixel resolution
	Must take images at both day and night time	Polar orbit (SSO noon/midnight)
Capable of measuring entire frequency range at all points over Finland	Payload capable of measuring RF radiation between 1 -10 MHz	1-10MHz region measured in 6 kHz strips. Sampling frequency 48 kHz. 50 samples from each 6 kHz band. Radio resolution 16 bits. AGC resolution 5 bits. Calculates summed value of the 50 data points of each frequency band.
	Adequate resolution for scientific measurements	
	Must compress data for faster downlinking	
Can communicate with ground station	Satellite sends and receives data via cubesat space protocol. Software includes scheduler.	Ground station uses Cubesat space protocol

The second requirement defines the operational requirement as:
"Suomi100 is a CubeSat."

This requirement defines on general that Suomi100 must meet requirements defined for CubeSat standard and other operational requirements like power consumption and downlink speed. For the interest of this thesis, going through them in detail is unnecessary. Only to note that Suomi100 meets the requirements set for CubeSat standard and requirements for power consumption and downlink speed are met.

2.5.2 Satellite operation modes

As the satellite has several operations it needs to perform, different *operation modes* were identified for the mission.

The operation modes are presented in detail below:

Target mode/measurement mode
The payload radio performs several sweeps over the entire frequency range. Because the orientation of the satellite has little effect on the payload radio antenna, the ADCS system is turned off. This is done to mitigate noise caused by the magnetorquers. The OBC calculates average values of the received signal power to reduce the size of the data. Alternatively, the raw data may also be stored in case the operator requests it. The satellite gathers telemetry atleast every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Low observation mode
This mode is similar to Target mode except that the payload radio takes measurements at a single frequency. This mode can be used to track ionosonde signals. The satellite gathers telemetry atleast every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Communication mode
Measurement and housekeeping data is sent down to the ground station via the Ultra-High frequency (UHF) link whenever possible, as data downlinking is the most restrictive factor of the mission. This mode is also used to send commands to the satellite. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Power charge mode
Only the essential components of the satellite are operating so that the solar panels can charge the satellite's batteries. Additionally, ADCS is used for optimal solar panel efficiency. housekeeping is gathered. The satellite gathers telemetry atleast every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Imaging mode
The onboard camera is used to take images of the earth, which requires the ADCS to accurately point the camera toward the earth. The images are either compressed by the camera or stored as raw images in the internal memory of the camera module. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Software update mode
Similar to the communication mode, the largest data traffic goes now up, with only the most essential telemetry being sent down. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded.

Idle mode (everything goes back to this mode)
The satellite always returns to this state if its not doing any of the other modes. The ADCS is off. The satellite gathers telemetry atleast every 10 minutes (should be prepared to go down to 1 minute) and sends a beacon every 1 minute.

Observation & imaging mode
The onboard camera is used to take images of the earth, which requires the ADCS to accurately point the camera toward the earth. The images are either compressed by the camera or stored as raw images in the internal memory of the camera module. The payload radio performs several sweeps over the entire frequency range before and/or after the image is taken. The satellite doesn't send a beacon during this mode, or gather telemetry unless specifically commanded. This is a data intensive mode!
Debug/status mode
This mode is specifically for checking out the satellite's health. Housekeeping can be gathered as quickly as 10 seconds (!), beacon is sent every 1 minutes, and all subsystems should be possible to be used. Use examples: e.g. timing of adcs turning, EPS solar panels functionality check, radio functionality check.
Deployment mode
The satellite starts in this mode - i.e. antennas are ready to deploy, 30 minutes switch-on time for EPS and 45 minute UHF radio beacon broadcast start time are ready start immediately when the satellite is deployed. The correct commands for the thermal knifes that cut the antenna lines are known and ready to start as soon as the EPS starts 30 minutes after deployment.

2.5.3 Instrument modes

In addition to the mission and operation modes, the different operational modes for the Suomi100 payloads were defined as well. For the radio payload, three different modes are defined. For the white light camera, one mode is defined. In addition, few macro modes containing both of the payloads are defined as well.

First mode for the radio instrument is tied to the *Low observation* operation mode. With this instrument mode, we use a single frequency to measure signals in the ionosphere. Table 2 shows the arguments related to this mode.

Table 2: My caption

Description	Values	Default
Mode starting time		"immediately"
Frequency	1-10 MHz	5 MHz
Number of measurements	>1	100 000
Skipped datapoints	>0	1000
Antenna	0/1	0

Like the first mode, the second mode for the radio instrument is closely related to the *Low observation mode*. In this mode we use a single frequency for the measurements, but individual measurements are not stored. Instead, certain statistical

values from a number of individual measurements are calculated and retained for analysis. These statistical values can be either (0) mean, (1) mean & median, (2) mean & median & standard deviation or (3) mean & median & standard deviation & minimum & maximum. Table 3 describes the arguments related to this mode.

Table 3: My caption

Description	Values	Default
Mode starting time		"immediately"
Frequency	1-10 MHz	5 MHz
Number of stored calculations	>0	100
How many measurements used in calculations	>0	100
Skipped datapoints	>0	1000
Which calculations performed	0,1,2,3	0
Antenna	0/1	0

The third mode for the radio instrument is similar to the second mode and tied to the *Target Mode* operation mode. In this instrument mode, we store statistical data about individual measurements like in mode two. But the frequency what we use is varied during the operation of the instrument. The frequency first starts at some value, measurements are made and stored and then the frequency is increased and measurements are made again. This procedure is performed until some defined maximum frequency is reached. Several cycles of this sort can be performed. Table 4 illustrates what arguments are part of this instrument mode.

For the camera payload, there is only one instrument mode defined. This mode defines which direction to point the camera, image quality and other parameters. In addition, this instrument mode is part of the *Imaging mode* operation mode. Table 5 describes these parameters in more detail.

By compining some of the instrument modes for the radio instrument and for the camera, several different macro modes can be constructed. For example, first performing the first mode for the radio instrument and then using the camera with its instrument mode and finally performing another measurement with radio instrument mode 3.

2.5.4 Functional system integration testing

The testing of Suomi100 satellite is performed for the purpose of (1) verifying subsystem integration and (2) satellite reliable operation as well as (3) verifying that

Table 4: My caption

Description	Values	Default
Mode starting time		"immediately"
Starting frequency	0.1-10 MHz	1 MHz
Ending frequency	0.1-10 MHz	10 MHz
Number of frequency values	>0	10
Number of cycles	>0	100
Skipped datapoints	>0	1000
How many measurements used in calculations	>0	100
Which calculations performed	0,1,2,3	0
Antenna	0/1	0

Table 5: My caption

Description	Values	Default
Mode starting time		"immediately"
Camera direction	1-6	1 (nadir)
Image format (0) RAW (1) BMP (2) JPEG	0-2	2
Exposure time	10000-100000	10 000 microseconds
Auto gain	0/1	0 (No autogain)
JPEG quality	0-100	85

the satellite meets its functional requirements. The functional requirements of the mission are described in sections 2.5.1, 2.5.2 and 2.5.3.

Features to be tested

Based from the research represented in section 2.1, the testing will focus on testing of features that have been thought of causing failures with CubeSats or that testing of them has been inadequate. In addition, tests will be performed for the two payloads

as well.

From the operational modes, four different conglomerates of features are identified for testing: (1) functionality of the camera payload, (2) functionality of the radio payload, (3) reliable operations of the basic software features of the satellite like housekeeping, safe reboots, software updates and so forth. Tests for the (4) "day in the life" operational scenario testing will be performed likewise.
Draw pretty pictures or draw them in section 4.

Approach for testing
Testing will focus on functional testing and it is performed at *System level* for all four features. Test automation is used in test execution and the tool for test automation is Robot Framework. The functional environment for each respective feature is to be simulated by inputs external to the satellite. For testing of the operation of the camera, natural light is used as an input. Testing of radio payload will use externally generated radio signals as input. The tests for "day in the life" scenarios will use a solar simulator and the satellite will be commanded over radio link. All these tests will be performed for the integrated satellite.

Test case Pass/Fail criteria
All tests are considered critical, thus a failure in execution of one test step in a test case leads to the test case to fail. Test steps are failed based on the responses of the satellite control software.

3 Methods

3.1 Suomi100 satellite

Suomi100 satellite is assembled together with a 1U Cubesat manufactured and designed by Gomspace company from Denmark. This 1U Cubesat, which is known as *NanoEye* in Gomspace product catalogue, forms the platform and main systems of the satellite [47]. This part of the satellite is referred as *platform* in this thesis henceforth. Picture of the platform is shown in Figure 10. On top of the platform, we added another payload, which consists of an *Amplitude Modulated* (AM) radio on a PC-104 type PCB and two ferrite antennas and a support structure. All designed an assembled in Aalto University by members of the Suomi100 satellite team. This part of the satellite is referred as *radio payload* in this thesis.

The subsystems and the satellite platform have flown several times in space aboard other missions. The platform forms a relatively well tested system with which we can investigate the development of automated functional tests for Cubesats. In addition, the radio payload and its control software in the platform give us another aspect for study. Namely, how to test the integration of a subsystem with the rest of the satellite.

The mission of the Suomi100 satellite is to take pictures of the northern hemisphere, especially of Finland. The satellite flies in the ionosphere in a polar orbit at an approximate altitude of 500 kilometers. With the radio payload a noise-map and natural noise levels in this area of the ionosphere could be devised.

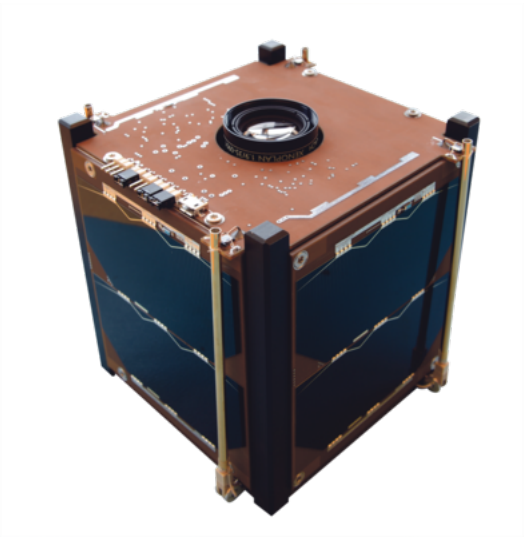

Figure 10: GomSpace NanoEye 1U. Courtesy of GomSpace A/S. [47]

3.1.1 Subsystems

The satellite consists of several subsystems. The central subsystem of any satellite is the system with the computer designated as OBC. In our platform it is known *NanoMind* and it is based on a Atmel 32-bit microcontroller [48]. Another vital system to the satellite is naturally the EPS and it is known as *NanoPower* in our platform [49]. Communication system of a satellite is the system responsible for receiving commands from the ground and responsible for sending information back to the ground as well. In our platform the communication system is known as *NanoCom* [50]. Besides these essential systems common to all satellites and spacecraft alike, we have as payload an optical white light wide angle Earth observing camera and the radio payload measuring AM frequencies. The camera came along with the GomSpace platform and is known as *NanoCam* in their catalogue [51]. Below the most essential subsystems to the topic of this thesis are desribed in more detail.

On-Board Computer - Nanomind

Figure 11: Nanomind OBC inside its casing. Courtesy of GomSpace A/S. [48]

The Nanomind A3200 On-Board-Computer shown in Figure 11, is based on Atmel AT32UC3C *Microcontroller unit* (MCU), which is a 32-bit *Reduced Instruction Set Computer* (RISC) microcontroller with advanced power saving features. This system runs the software that is responsible for the majority of operations of the satellite and it works as sort of a mediator between subsystems and routes communication between them correctly. The software is explained in more detail in the following subsection. The MCU has two *Inter-Integrated Circuit* (I2C) buses and one *Controller Area Network* (CAN) bus for communication with other subsystems. It has also 8 ADC pins, which can also be programmed to work as *General Purpose Input-Output* (GPIO) pins. Nanomind contains a *Synchronous Dynamic Random Access Memory* (SDRAM) with 32 MB of capacity for volatile storage as well. For non-volatile storage, the subsystem has a 128 MB NOR Flash. Below in Figure 12 is a block diagram of the OBC. [48]

Electrical Power System - NanoPower

The Nanopower P31 on our satellite contains two lithium-ion batteries and has several reliability features. Figure 13 presents how the subsystem looks. The batteries

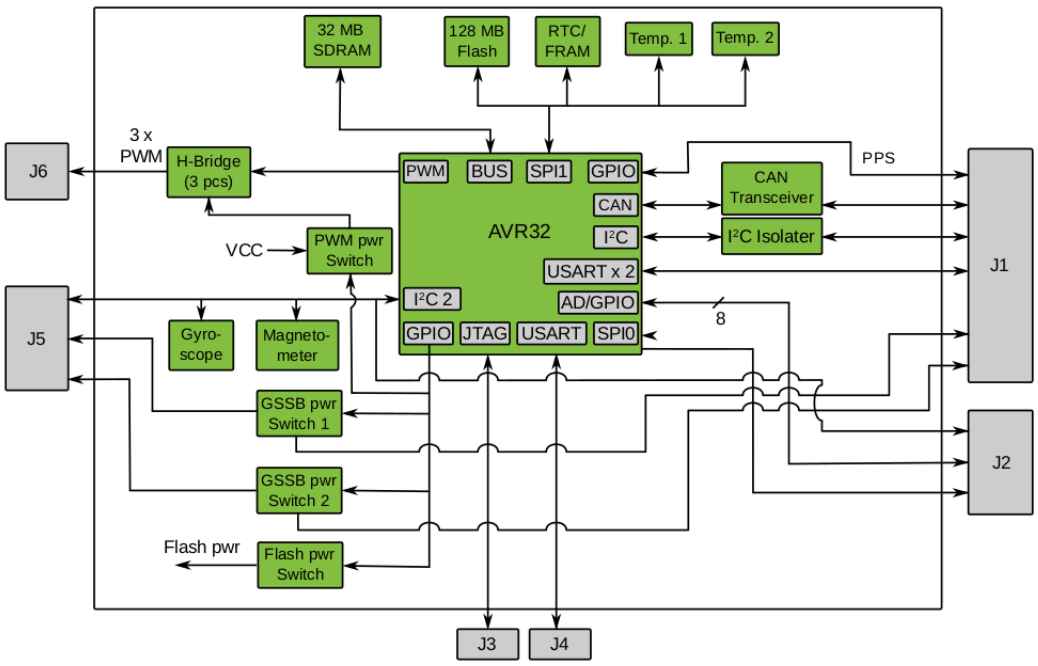

Figure 12: Block diagram of Nanomind. Courtesy of GomSpace A/S. [48]

Figure 13: Nanopower EPS. Courtesy of GomSpace A/S. [49]

are charged by the five solar panels aboard the satellite and these then provide power to the whole satellite through the stack connector on the PCB of the EPS subsystem. The system has its own microcontroller, which measures the voltages, currents and temperatures of the system. The microcontroller can also be used to control the 5 V and 3.3 V power buses of the EPS among other features. [49]

Communication subsystem - NanoCom

The NanoCom COMM system shown in 14 is a software configurable half-duplex transceiver designed for long-range transmissions. Certain parameters of the system can reconfigured on-orbit. These parameters are frequncy, bitrate, modulation type and filter-bandwidth. Data rates can be between 0.1 - 115.2 kb/s. The subsystem has its own microcontroller as well as essential radio elements such as *Power Amplifier*

Figure 14: NanoCom communication system. Courtesy of GomSpace A/S. [50]

(PA) and *Low-noise amplifier* (LNA). [50]

Camera payload - NanoCam

Figure 15: NanoCam payload camera. Courtesy of GomSpace A/S. [51]

First of the payloads in Suomi100 satellite is the NanoCam wide-angle white light camera, presented in Figure 15. The subsystem consists of a lens, image acquisition and processing board. The lens is an industrial grade lens and the image acquisition element is an *Aptina MT9T031* 3-megapixel *Complementary Metal Oxide Semiconductor* (CMOS) color sensor. The processing element consists of a PCB with components such as an *Atmel SAMA5D35* processor with clock rate of 536 MHz, 512 MB of DDR2 memory for image storing and processing and of a 4 GB eMMC flash drive with 2 GB for image storing. [51]

The software for image processing and storing runs on a customized embedded *Linux* (GomSpace Linux) and there are several features for image acquisition and storing. The images can be stored in either *RAW, BMP* or *JPEG* formats. Several parameters of the camera system like exposure time, different gain values, gamma correction and so forth can be altered on-orbit. [51]

Radio payload

The second payload of Suomi100 satellite is the AM radio payload. As noted, this payload was developed by the Suomi100 satellite team, namely by M.Sc Petri Koskimaa, B.Sc Amin Modabberian and B.Sc Arno Alho. Figure 16 shows the PCB of this subsystem. Central to the system is the PCB with a *Integrated Circuit*

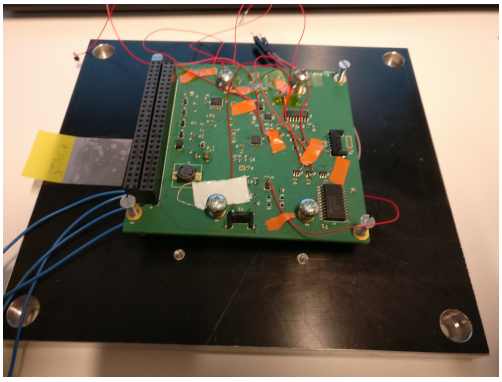

Figure 16: Protomodel of the AM radio payload PCB.

(IC) with model *Silicon Labs Si4740* automotive *Amplitude Modulated/Frequency Modulated* (AM/FM) Radio receiver [52]. It can receive signals with frequencies from 149 kHz to 23 MHz in 1 kHz steps. The Si4740 can be set to receive AM, AM/SW/LW or FM signals. Several features of the IC can be modified. Including frequency, volume, output format, sample rate, attack rate, release rate and many more. Commands to the Si4740 are sent via the I2C bus and the output of the receiver is read via the SPI bus [53].

Another important element of this subsystem are the antennas and their support structure. The antennas were designed by M.Sc Petri Koskimaa. Design and construction of the antennas are described in his Master's thesis, titled "Ferrite Rod Antenna in a Nanosatellite Medium and High Frequency Radio" [54]. These antennas are four ferrite rods, with two on either side of the support structure forming one antenna. The other antenna is used when listening to frequencies below 2 MHz and the other is for frequencies between 1.0 and 9.3 MHz.

The support structure for the antennas was developed by Ph.D Antti Kestilä and it was made with a 3D printer using material that can sustain the environment in space.

3.1.2 Gomspace software

Besides the subsystems for the NanoEye platform, GomSpace also provided software for all these subsystems. The essential core of the software architecture is a delivery protocol known as *CubeSat Space Protocol* (CSP), which was originally developed in 2008 by a group of students from Aalborg University in Denmark. The protocol has further been developed and maintained by GomSpace itself. In practice, the protocol is used for communication between different subsystems as well as with the ground station. [55]

The protocol as well as the software for subsystems were written in *C* programming language. In addition to the specific software for each subsystem, all the systems share a set of common functionalities. These common functionalities include sending and storing of *housekeeping* (HK) data, parameter tables for adjusting different functionalities of a given subsystem, logging functions and inter-subsystem

communication through CSP. In addition, each subsystem provides a terminal shell known as GomSpace Shell or *GOSH* for control of the subsystem via a PC using *Minicom* software. [55]

The software developed by GomSpace for NanoEye also includes such general functionalities as *File Transfer Protocol* (FTP) running over CSP, with which files and data can uploaded and downloaded from the satellite as well as some basic file handling routines can be handled with the FTP. Among the file handling functionalities is the ability to compress or decompress files with the *ZIP* format. The software in the satellite can also be updated via the FTP by uloading a software image to the satellite and telling the computer to start reading from it after next reboot. In addition to these, *Flight Planner* is another general feature and with it commands can be set to execute at certain points in time either once or repeately with some interval. [55]

The operating system running in the NanoMind OBC is a free *Real-time Operating System* (RTOS) known as *FreeRTOS*, which is a light weight operating system designed for embedded systems that use microcontrollers and small microproserssors [56]. It was developed by *Real Time Engineers Ltd.* in USA. The version of the operating system used in our satellite is 8.0 at least. The operating system is mostly written in C programming language, but certain necessary parts are written with *Assembly* programming language.

FreeRTOS is a real-time scheduler where different tasks execute in a *Round Robin* fashion, where each task is given some priority value and tasks with higher priority value are given more processing time and those with the same priority value take turns in execution of instructions. FreeRTOS offers three different variations of this scheduling policy and for NanoMind we have one of...FreeRTOS manual sivu 118. In addition to scheduling, the operating system offers functionalities for inter-task communication via semaphores at least. [56]

gs-a3200-sdk-v1.2.pdf About CSP. CSP client and commands. FreeRTOS.

3.1.3 Satellite control software - CSP Client

The ground station software used to control the satellite is known as *CSP Client*, which is a simple console program for remotely sending commands to the satellite via CSP. The program itself is written in C programming language. The syntax of the software is almost identical to the Gomspace Shell found in the subsystems manufactured by GomSpace. As the source code was available to us, we were able add our own commands to control our radio payload among other things. In Figure 17 the CSP client is shown running in *Debian 9* Linux, showing for example a command inquiring for housekeeping data from EPS subsystem.

The software has over a hundred commands if the subcommands related to the main commands are counted. Thus only the main commands used in test automation of Suomi100 are presented here:

reboot <CSP node>
Explain.

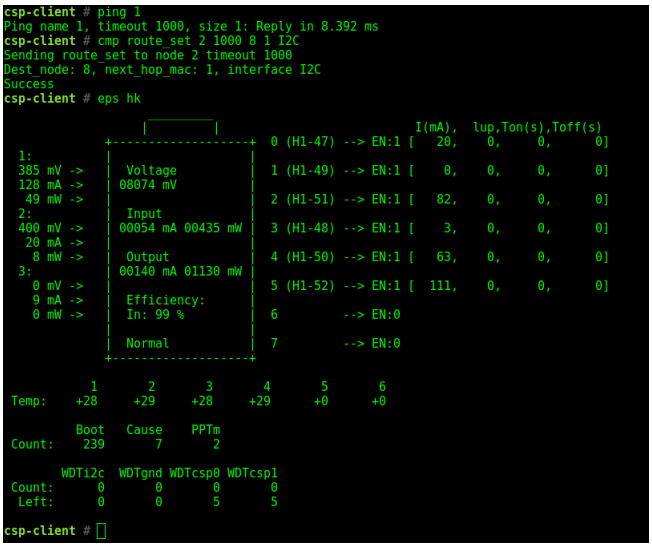

Figure 17: Suomi100 ground station control software running in Linux.

shutdown <CSP node>
Explain.
hk get <type> <interval> <count> <t0> <path>
Explain.
eps hk
Explain.
ping <CSP node> <timeout>
Explain.
rparam download <CSP node> <mem>
Explain.
rparam set <name> <value>
Explain.
rparam get <name>
Explain.
rparam send
Explain.

3.1.4 Software for radio payload

The software for controlling the AM radio instrument was developed by B.Sc Juha-Matti Lukkari, the author of this thesis, and by M.Sc Jouni Rynö from *Finnish Meteorological Institution*. Unlike in most of the subsystems in Suomi100, the radio payload doesn't have its own microcontroller or any other general purpose computer. Therefore, the control software operates as few FreeRTOS tasks in the NanoMind. In addition, new commands for operating the payload instrument were added to the CSP client as well as to the NanoMind GOSH terminal.

One of the tasks receives a command as a CSP packet, which is then parsed as a

command to be sent via the I2C bus on NanoMind to the Si4740 IC for example. Commanding of the Si4740 is based on the hexadecimal values of the bytes it receives [53]. The first byte received defines which action is being performed and the following bytes define the arguments for that respective action [53]. The IC then gives a respond byte, with hexadecimals 80 and 81 implying a succesful command and for example 40 or 0 implying a failed command [53].

Over 50 different arguments for different commands can be used when controlling the Si4740 [53]. Therefore, when commanding the radio payload to perform a measurement, the different values for different values are loaded either from a configuration file or from a GomSpace paramater table. The parameter table and the configuration file were added to NanoMind by us. All the commands can choose to use either one of these commands and in addition the commands can be used "manually" without loading any external configuration for the command.

Some of the most essential commands for the radio payload operation in CSP client are presented here:

radio on <config> <reg>
Explain.
radio operation <config> <config> <mode> <mode arguments>
Explain.
radio set_property <config> <property> <value>
Explain.

3.2 Automation of control and testing of the satellite

3.2.1 API and communication layers for CSP Client software

In order to automate the control of the satellite, we need some form of interface which can communicate between the satellite and the framework used to perform the tests. Fortunately, the Gomspace software already provides a terminal shell program called *GOSH* on each of the subsystems [48]. In addition, all of the subsystems can be controlled from a single shell via a serial to USB FTDI cable connected to NanoUtil USB port [57]. As presented in previous subsection, a separate CSP client software exists, which can be used to control the satellite from the groundstation via a radio link and it can also be used to control satellite via the FTDI cable.

Automating the control of the CSP client software was chosen as the solution on how to automate the control of the satellite. The CSP client was chosen, because by automating control of it, we can do tests via the radio link as well. The automation was first done by modifying the source code of the *main.c* file of the CSP client, which contains the C-language main function for the program. The modification consists of creation of a POSIX thread (pthread) which runs a function that opens and listens to a socket connection on the localhost network address. When a message is received on the opened socket, the thread then runs the command on the CSP client terminal, as if a user would have written the command on the terminal. Alternative solutions

could have been used, for example a separate program could have been written and the CSP client could have communicated with it through some of the inter-process communication methods provided by Linux operating system. This could have been made through Linux output and input *standard stream* redirection methods like *pipes* [58]. Using the network connection on the localhost has the benefit of potentially being externally controllable.

Over the course of development of the libraries and test suites, a more direct approach of using the *Standard Input Stream* (Stdin), to send commands to the CSP client was also developed. Furthermore, an even more direct method of simply automating the keypresses of the keyboard was implemented with the aid of a Python library called *pyautogui*. The benefit of having the communication performed with Stdin or with automated keypressing through Linux kernel keyboard driver is that we can automate the use of not just our CSP client but the use many different terminal programs that are run locally on the computer running the tests. Even programs with source codes that we have no access to, thus omitting the need to write a separate *Application Programming Interface* (API) into them as was done in our case. Nonetheless, using self-tailored process communication APIs via e.g. pipes or sockets have some advantages over these sort of "crude" methods. For example, use of Stdin can be reserved to the program in a way that it is not accesible outside the program itself. Sending the commands by automating keypresses can bypass this. But if for example something else is done with the computer during testing, the keypresses are received by programs that we didn't wish to automate.

Nonetheless, in order to create a generic test automation library, all of these methods of process communication were later in development incorporated to the release version of the CubeSat test automation library, which is explained in more detail in section 4. If we write the terminal software on our own with our testing library in mind, all of these communication methods should be valid for automating the testing.

Besides requiring the method of how to send commands to the satellite in an automated fashion, we also require to know how the satellite responds to these commands in order to verify the tests as either passed or failed. The CSP client software fortunately receives responds to the commands sent to the satellite and thus we have some knowledge of how did the satellite behave. It was found that the easiest solution would be to read the *Standard Output Stream* (Stdout) of the CSP client process and therefore transfer the responds to our automated verification functions.

Other way to the capture responds to the commands would be to modify the source code of the CSP client for it to send the received outputs of the executed commands to another port on the socket connection. In this way we could then listen to this port on our Python test library. Doing the transmission of CSP client output to the test automation libraries this way was experimented by the use of some Linux output redirection routines like *dup*. However, there were some difficulties with the implementation and due to time constraints it was easier to monitor the standard output of the client software. Furthermore as mentioned before, during the development of the test libraries, use of Stdin for communication was developed

as well. In fact, as with using Stdin to send commands to the process, reading the Stdout of the process allows us to generate a generic test verification solution to this as well. Provided that the process which we wish to do automated tests with responds through the Stdout stream, which fortunately happens to be the case for most terminal programs [source?].

The solution for the communication is illustrated in Figure 18 and the modified main.c for the CSP client can be found in the Appendix.

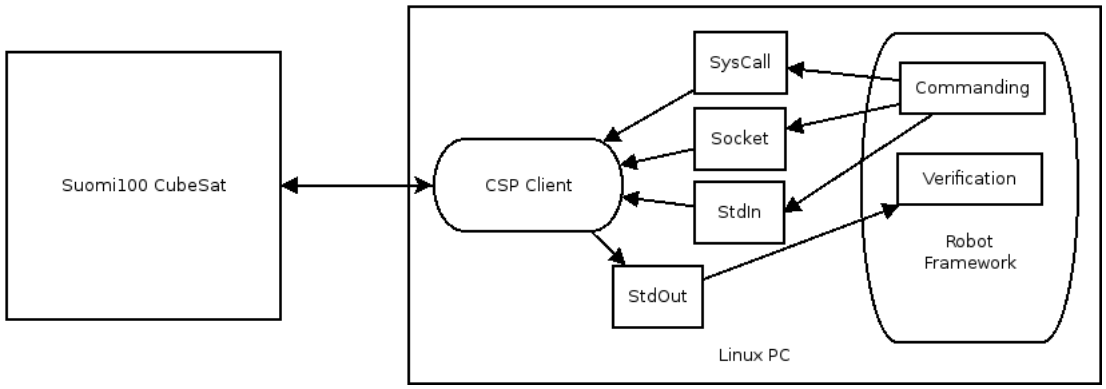

Figure 18: Illustration of the software architecture developed for Suomi100 test automation. Large rectangles represent environments, small ones represent layers and rounded rectangles represent programs.

3.2.2 Python libraries

A set of scripts using Python programming language were written, called libraries in the context of Robot Framework. All of these libraries consisted of one Python class each. The class of the core library, known as *CubeSatAutomation*, has the methods for the communication with the CSP client via any of the three methods, socket, Stdin or automated keypressing via pyautogui library. Furthermore the library includes the methods to read and verify the process replies from Stdout. As can be seen in Figure 18, the commands are sent to CSP client through any of the three communication routes and the output of the program goes to the Stdout. The output is then caught and read by the CubeSatAutomation library and test cases and test steps or keywords are failed or passed based on the output read from the CSP client.

It is importation to note the following details about the test automation libraries.

CubeSatAutomation library

CubeSatAutomation library has the core functions for sending commands, opening socket connection, opening and closing the opened program and others. Besides being able to open the CSP client program, any program can be automatically opened with the library as the method for opening uses the standard Python *subprocess* library. All the other libraries implemented use the core methods in CubeSatAutomation, for example, to send commands to the CSP client to execute. These other libraries have subsystem specific functions for the test automation. To create only one open

communication route between CSP client and Robot Framework and to only one handle on the CSP client program process, the core library defines these as class variables, which are then accessed by the subsystem libraries. In practice this means that we don't open several CSP client programs and several connection routes separately for each subsystem library included in the test suite. Instead, the program and the communication route is opened only once for each test suite. Another class variable which defines the scope of the instance of the class in the Robot Framework was defined as well. This was chosen to have a value corresponding to the suite level, as the widely used *Selenium2Library* uses the same scope for its class instance in Robot Framework as well. By having the scope on the suite level, only one instance of the library class is declared per test suite, thus again having only one handle on the client software and having only one connection route open during the execution of a test suite.

The essential core methods used in the CubeSatAutomation Python class are explained below:

Client Start <config file> <program> <parameters
Explain.
Client Close <socket> <program>
Explain.
Connect Socket <config file> <server> <port>
Explain.
Send Command <message> <option> <timeout> <read timeout>
Explain.
Write Command <message> <option> <timeout> <read timeout>
Explain.
Type Command <message> <option> <timeout> <read timeout>
Explain.
Persistent Command <message> <exception replies> <end reply> <timeout> <read timeout>
Explain.
Verify Reply Contains <message> <timeout> <read timeout>
Explain.
Verify Reply Contains Not <message> <timeout> <read timeout>
Explain.
Verify Reply Contained <message> <timeout> <read timeout>
Explain.
Wait Until Reply Contains <message> <timeout> <read timeout>
Explain.

Finally, here are some of the keywords presented that are specific to GomSpace NanoEye.

Set Satellite Parameter <device> <parameter> <value>
Explain.
Send Satellite Parameters
Explain.

Remember to mention that replies can be stored temporarily.

Creation of a skeleton core library was the aim of the final development of our test automation library. This library only has the aforementioned methods to start and communicate with a desired Linux program running on terminal shell (or bash shell) and keywords that are more specific to Suomi100 or CSP client were omitted. With the aid of this library, satellite software developers wishing to automate testing of their satellite and satellite software, can then create their own specific libraries suited to their own needs. The final version of the CubeSat test automation library is desribed in section 4.

Subsystem libraries

Other libraries developed for test automation of Suomi100 are called *NanoCam.py* and *RadioPayload.py* and these were intended for the automated testing of NanoCam and radio payload subsystems respectively. The classes of both of these create an instance of CubeSatAutomation class instead of calling for specific functions or methods of that class. By doing this the class variables including handle to the automated process, socket address and others are passed to all these other classes as well. The methods of these classes thus use the methods of CubeSatAutomation directly.

The specific methods defined by the NanoCam test automation library are presented below as Robot Framework keywords:

Camera Startup <timeout>

Reboots the camera and downloads parameter table 1 from the subsystem. Timeout specifies the time that we wait for the camera subsystem to come online in satellite bus.

Camera Take Picture <timeout> <store format> <filename> <autogain>

Sets image format and filename in the camera and takes a picture with the given autogain value (empty at default). Keyword fails if the image is too dark (less 5 % light) or too bright (over 95 % light).

Camera Load Picture <stored file> <loaded file>

Downloads the file stored in NanoCam to the PC running CSP client program.

The subsystem specific keywords for the radio payload are defined as the following:

Radio Startup <switch input> <switch power> <antenna input>

Explain.

Radio Powerdown

Explain.

Verify Radio Status

Explain.

Run Radio Mode <parameter file> <property file> <mode> <mode arguments

Explain.

Verify Radio Results <buffer file> <timeout>

Explain.

Radio Load Data <stored file> <loaded file> <timeout>

Explain.
Radio Plot Data <file> <output file> <plot image>
Explain.

– Libraries are Static API version

3.2.3 Robot framework test suites

The test cases follow the keyword-driven approach and the keywords are written to be short and mostly to be non-specific to the test case. The functions and methods written in Python and described in previous section are used directly as such. Because the approach was to make smaller set of versatile and generic keywords that could be used over many test cases and test suites. This approach was felt to be more efficient as there would be less need to maintain the test suites if they had large set of specific, though descriptive, keywords. Besides, our satellite project is not a typical software project where stakeholders would go over the test suites and validate them. All people involved in the project have a technical background. In addition, having a set of general keywords that are not entirely tied to our satellite project is beneficial if some future satellite project wishes to use the testing methods and tools described in this thesis.

Each of the test cases are tied to a particular operation mode. The operation modes are discussed in detail in section 2.5. Purpose of each test case is to verify functionality of some aspect of the particular operation mode. Each test case is marked with the Robot Framework *[Tags]* marker to identify which operation mode the test case is related to. The test suites are divided firstly based on the four different larger features what we are testing. Namely, separate sets of test suites are written for camera payload, radio payload, NanoEye basic functionality and for "Day in the life" testing. Each aspect of a feature further divides the test suites into test suites testing different parts of the particular feature.

3.3 Test setups and environment simulation

As defined in section 2.5, different aggregates for testing were identified from Suomi100 satellite and for simulating the functional environment of the satellite for these different types of tests, four different environments were set up. Two different environments for testing two different payloads, one for testing basic operational features of the NanoEye platform and one larger for the operational scenario testing of the satellite.

3.3.1 Environment and setup for camera payload testing

For the testing of the NanoCam and imaging operation mode, we tried to find something facing the camera with equal colour and brightness values as what the camera would see while in orbit. The easiest solution is to simply take the whole integrated satellite outside on a bright day to the balcony on top of our department in Aalto University. The satellite is standing on top of a stand and the side with

the camera lense is directed towards horizon. A PC with CSP client and the test automation tools are connected to the satellite via the USB connection on NanoUtil. In addition, the satellite is loosely enclosed in a plastic container to protect it from particles in the air.

Below is Figure 19 of the test setup used during the testing.

Figure 19: Suomi100 satellite on a balcony during imaging mode tests.

3.3.2 Environment and setup for radio payload testing

Environment for the testing of the radio payload is set up in the clean room of Aalto University's space laboratory. The satellite is connected via the USB connection to a PC with the CSP client software and the test automation tools. The functional environment was simulated with a radio signal source in order to create some artificial noise in radio frequencies that would mimic the radio signals present in the ionosphere. The radio noise is generated with a *HackRF One* Software Defined Radio (SDR) which is connected to another PC running *GNURadio* signal processing software.

Figure 20 shows the setup for the testing of the radio payload.

The frequencies that are used in the environment simulation are 2 MHz, 5 MHz and 9 MHz. These were chosen based on the requirements of the payload (should operate in range of 1-10 MHz) and the limitations of the hardware, as the HackRF can't produce signals with lower frequencies than 2 MHz. Furthermore, the antennas attached to the payload themselves cannot receive signals that are much higher than 9 MHz. The middle frequency was chosen to be 5 MHz as based on the research done on the radio signals in the ionosphere, this frequency would be of most interest to us [59].

Figure 21 shows *Fast Fourier Transform* (FFT) plot of the noise that is generated from the GNURadio, which is then transformed into radio waves by the HackRF.

Figure 20: Radio payload testing with *HackRF One*.

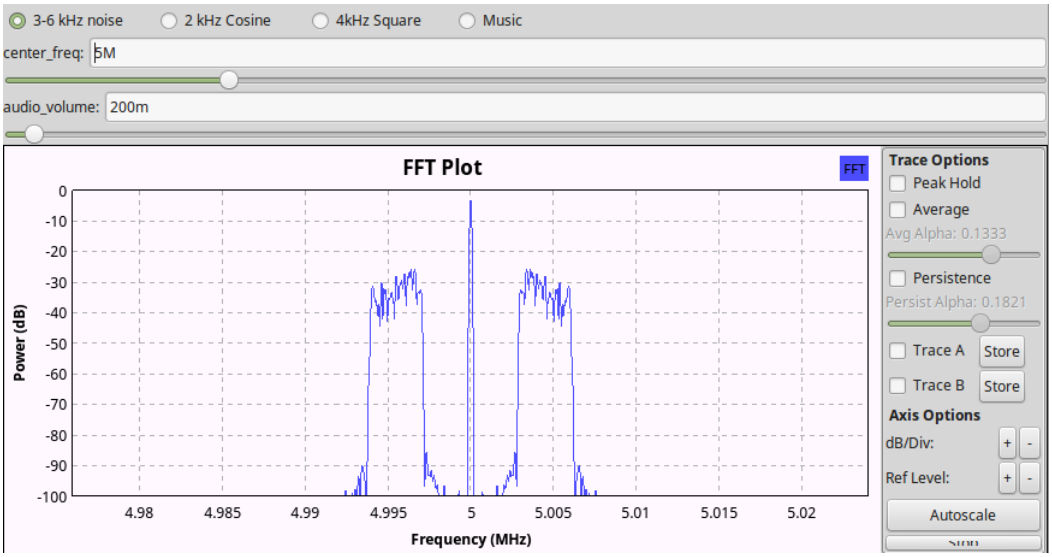

Figure 21: Screenshot from *GNURadio* signal processing software showing FFT plot of a radio noise being generated.

3.3.3 Environment and setup for satellite basic operations testing

For testing of the basic satellite operational features like collection of housekeeping and safe rebooting during an error, no external inputs to the satellite are used. The satellite is in the clean room of Aalto University's space laboratory connected to a PC with the CSP client. In Figure 22 we have a picture of this setup.

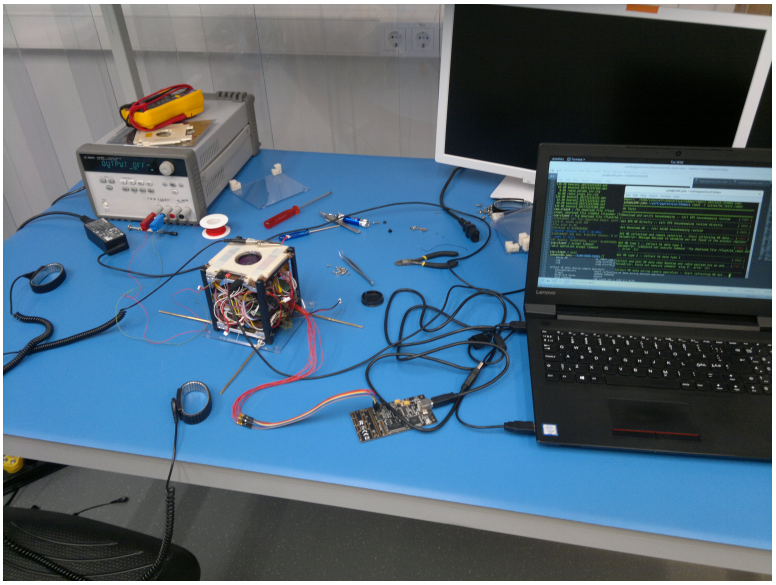

Figure 22: Setup for testing of basic functionalities of the NanoEye platform. Suomi100 is on the left and a PC with CSP client and test automation softwares is on the right.

3.3.4 Environment and setup for satellite operational scenario testing

In "day in the life" operational tests, the Sun is simulated with a 1800 Watt Xenon lamp that is situationed approximately 1.5 meters away from the satellite. Two solar panels are connected to the satellite in the manner they are connected during flight. The satellite faces the lamp in an angle so that both panels receive light from the lamp. As the lamp is quite powerful, we can really verify that the solar panels charge the batteries in the satellite. In addition, the lamp can heat the objects it is faced towards and this was used as a method to add some thermal features to the test. The idea is to use the lamp for the time it takes the satellite to heat up to 50 degrees celcius under the illumination and to then let it cool back to room temperature (approx. 27 in our clean room). The heating and cooling was measured with *FLIR E6* thermal camera and the heating time was measured to be approximately 10 minutes and the cooling down period was measured to last ca. 20 minutes.

In Figure 23 this setup with the solar simulator is presented.

For the "day in the life" testing, these periods form the phases of the operational scenarios. When we pretend that the satellite comes from eclipse, we turn on the Xenon lamp and we are in communication with the satellite for 10 minutes and

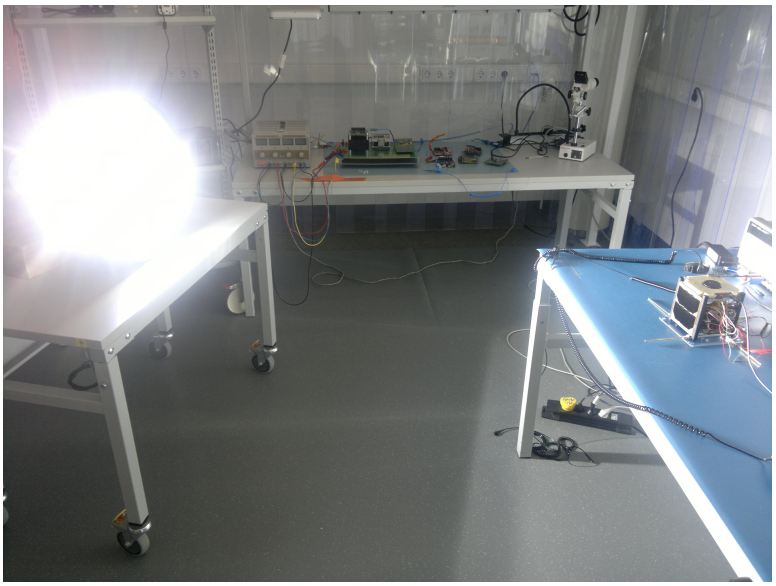

Figure 23: Day in the life setup for the satellite. The Xenon lamp is on the left and Suomi100 CubeSat is on the right in the picture.

after that the lamp is turned off and we pretend that the satellite goes out of the reach of our ground stations and stays there for 20 minutes. In addition, the 10 minutes communication window roughly corresponds to the time that we can be in communication with the satellite during one revolution around Earth by the satellite.

Unlike in the setups described previously, the control of the satellite happens via radio link. A SDR with model *Ettus USRP B200* is connected to a PC with the CSP client software and the test automation softwares. This SDR is the one used in the actual ground station and for this testing the SDR and the PC were located in the next room in Aalto University's space laboratory. The actual groundstation and all of its hardware is not used because the solar simulator has to be controlled manually and the groundstation is situated several floors up from the Aalto University's Space laboratory.

Figure 24 presents this setup.

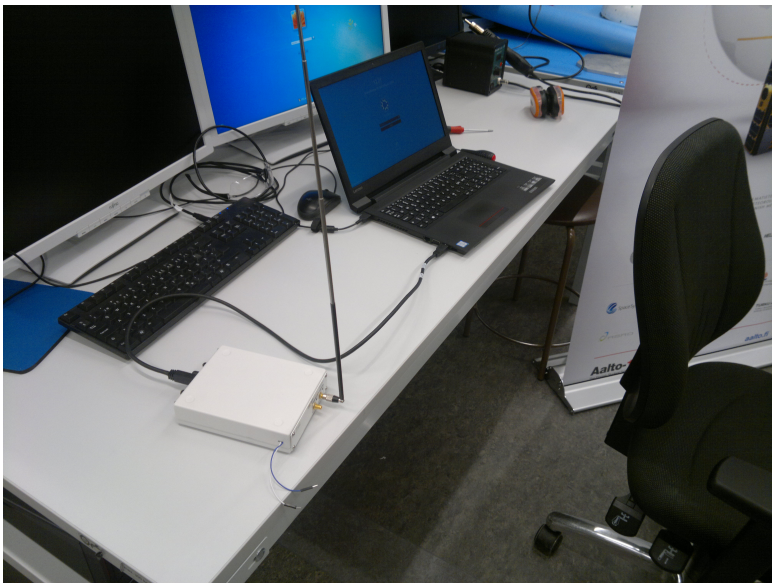

Figure 24: Day in the life setup for a "stripped down" groundstation.

4 Results and Discussion

4.1 Executed tests

The Robot Framework test suites executed can be divided into three categories: test suites done for the two payloads (1), test suites written for general satellite operations (2) and test suites for "day in the life of the satellite" operational scenario tests (3).

The first category of suites follow more traditional method of combinatioral testing. Each test case was derived from the operation modes defined for the camera and radio payloads. Each of the test cases for their representative operation mode are identical in steps, but use different combination of values in the keyword arguments.

The second category consists of test suites made for the functional testing of the commands that contribute to the basic operations of the satellite. These test cases for different operations are completely different from each other and the tests here begin to resemble different scenarios or use cases. Such as telemetry gathering, flight planner commands, software update, etc. In addition, test suite was written for testing restarts of the subsystems and of the OBC and of the whole satellite as well.

The third category, follows the higher level satellite integration tests where we test the satellite based on operational scenarios. Separate test suites were written for each operational scenario. The test cases at this level are simply different phases in the operational scenario.

In addition to these test suites, another one was used during the development of the software for the radio payload. This test suite resembles a smoke test and it has limited set of test cases, testing each of the operation modes. Yet, no proper continous integration chain was done during this time, but the smoke tests were manually started usually once every week.

4.1.1 Tests for camera payload

The test cases for the NanoCam were identical in structure, or in other words, we made several test cases for the same use case of the camera payload. Only difference was that the main parameters like gain value and exposure time differed between test cases. These two are the main parameters according to the NanoCam manual provided GomSpace [51]. Having the exposure time fixed, we demonstrated the use of combinatorial testing to go through different variations of the camera parameters. Three different values of the exposure time were used and for each value, the same parameters other than exposure time were changed in different test cases. The values used for the exposure time were $10ms$, $30ms$ and $90ms$. Below is Figure 25 showing the test case structure used in testing of the camera payload.

As can be seen from the example test case, the test covers such features as NanoCam restart and detection in the satellite bus, the use of different camera parameters and finally, image taking, storing and transfer from the satellite. All taken images were then added to the Robot Framework log files, which gives a comprehensive view on how the different camera parameter values affected the taken images.

```
Imaging mode - Exposure 10000 Gain-Target 60
        [Documentation]                 The onboard camera is used to take images of the earth.
        [Tags]                          OPMODE-IMAGING
        Satellite State                 Idle
        Camera Startup                  15
        Verify Startup                  Camera
        Verify Device Detected          Camera          5
        Set Satellite Parameter         Camera          exposure-us                     10000
        Set Satellite Parameter         Camera          gain-target                     60
        Set Satellite Parameter         Camera          gain-global                     2048
        Set Satellite Parameter         Camera          jpeg-qual                       85
        Set Satellite Parameter         Camera          color-correct           true
        Set Satellite Parameter         Camera          gamma-correct           true
        Set Satellite Parameter         Camera          white-balance           false
        Send Satellite Parameters
        Camera Take Picture             5000            2               def.jpg         -a
        Camera Load Picture             /mnt/data/images/def.jpg                def1.jpg
        Log                             <img src="def1.jpg" width="500" height="500">   html=yes
```

Figure 25: Robot Framework test case structure for camera payload testing. The 1st column represents the command or action performed in CSP client (see section 3.2.2-3.3.3), second, third and fourth columns define parameter options and values for the commands.

For the environment simulation, the attempt was to find a really sunny day to give some indication of the brightness of the pictures while the satellite is in orbit. This was achieved to some extent, but at the start of the test few clouds appeared and some pictures turned out very bright and some less so. If the level of light would have stayed the same during the whole test, we could have gotten some baseline information about the affects of the different parameters on the images.

Nonetheless, no test cases failed due to software errors. These tests on the camera provided certain confidence that changing different parameters didn't crash the software or that any combination of parameters didn't cause problems in the functioning of the satellite. We also observed that the images didn't become distorted in any way. Out of the 39 test cases, 9 failed because the light level was too high. Yet, the brightness in space around Earth is much higher than what we can have here on the surface even at the brightest day [source], not to mention the fact that the tests were conducted in Finland. Thus, increasing the exposure time or gain value while in orbit will make the images to be too bright and therefore, setting the camera parameters to their default values would possibly give us the best pictures. One should recall that, the automation of these tests was the first time our automation libraries along with Robot Framework were used to properly test Suomi100 satellite and therefore these tests worked as a technology demonstration as well. Figures 26 and 27 show some pictures taken with the NanoCam camera on the Suomi100 satellite.

Most importantly, the tests demonstrated that the integration with the rest of the satellite was succesful. As in fact, there was a defect in the integration of the camera with the satellite at the first time the satellite platform arrived to us. The camera lense was too far away from the cell in the PCB of the camera subsystem and the first test pictures taken with the camera were distorted. The manufacturer of the satellite platform provided us with a test picture that the camera was working

properly, but it seems that they didn't test the camera after it was integrated to the satellite. After this problem was found, GomSpace provided us with a new camera and the integration of this worked properly. The Robot Framework tests worked thus also as the verification tests for the integration of the NanoCam subsystem to the satellite platform.

Figure 26: Picture taken at Maarintie 8, Espoo, with NanoCam integrated on the Suomi100 satellite. Camera parameters were set to the default values provided by NanoCam manual [51].

4.1.2 Tests for radio payload

During the development of the software for the payload radio, a small test suite was used for smoke testing of the software and the payload. Basically, the test cases tested that the general commands are executed without errors and that the radio can output data. This test suite was also used when the payload was integrated to the satellite.

The three tests in this suite were run at least once a week during the development of the software, but a proper *Continuous Integration/Continuous Development* (CI/CD) pipeline was not used where for example flashing of a new software to the NanoMind would have caused these test to run automatically.

Besides the aforementioned smoke test, a more comprehensive set of test suites was performed for the radio payload after it was confidently integrated to the satellite

Figure 27: Picture taken at Maarintie 8, Espoo, with NanoCam integrated on the Suomi100 satellite. Camera parameters were set to use exposure time of 30 milliseconds and to use no gamma correction.

platform. As the hardware and software were of our own design and the system integration occurred with a platform manufactured by another organization, these tests took the most time of all automated tests done on Suomi100 satellite. A few sessions were held where we ran these comprehensive test suites for the radio payload and each time new defects in the code and in the integration were found. Of all the tasks related to Suomi100, the proper integration of the radio payload to the GomSpace 1U NanoEye took the most effort from us.

Test cases for our payload followed the combinatorial testing method in a way where all the the test suites for a given radio operation mode had the same test cases but the frequency used was different. In addition, the test suites for different radio operation modes differed as each had a bit different parameters. These test suites then covered some amount of different parameter combinations and as with the camera payload, the measurement data was downloaded from the satellite and then processed and plotted and the figures were added to the Robot Framework HTML log files.

In Figure 28 is an example of the test case structure used in testing of the radio payload.

Most interesting information about the payload operation was found from the

```
Lowobs Mode - 5 Mhz Default parameters
        [Documentation]         The payload radio performs several sweeps over the entire frequency range.
        [Tags]                  OPMODE-LOWOBS
        Satellite State         Reboot
        Radio Startup           3       0       1
        Verify Startup          Radio
        Verify Device Detected  Radio   5
        Verify Radio Status
        Store Client Responses  Lowobs Mode              80              15
        Run Radio Mode          /flash/radio_params.cfg  /flash/radio_props.cfg  2  0;5000;100;100;100;0;0;
        Sleep                   2
        Get HK                  30  2  1  1  5  2  /flash/hk_test_lom
        Send Beacon             10  4  1
        Sleep                   10
        Verify Radio Results    Lowobs Mode              80
        Radio Power Down
        Radio Load Data         /flash/data/m2_debug.dat        m2_debug1.dat
        Radio Plot Data         m2_debug1.dat                           m2_debug1.txt    m2_debug1.png
        Log                     <img src="m2_debug1.png" width="640" height="480">      html=yes
```

Figure 28: Robot Framework test case structure for radio payload testing.

CSP client replies outputted to the log files. What we were measuring was static and as such, nothing too much could have been said about the test cases just by looking the measurement plots. Besides the fact that the values were not zero or that there was actual variance in the values measured. In Figure 29 below we have one plot produced from measurement made with the radio while the satellite was at our laboratory at Aalto University and no external radio signal sources were generated by us.

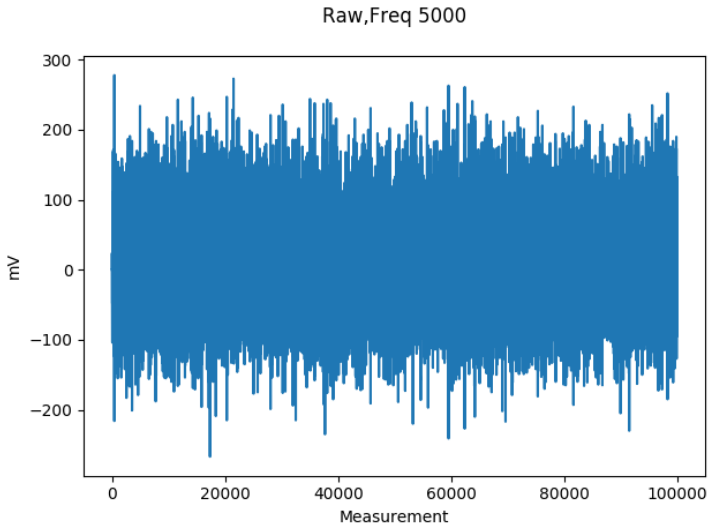

Figure 29: Plot of measurements of radio static made by the radio payload at 5 MHz.

During integration and testing of the radio payload, several problems occurred. Some of these problems can be attributed to be *emergent problems in integration*. For one, the payload was developed so that a *RaspberryPi 3* computer simulated the OBC. This computer has a Quad Core 1.2 GHz processor from Broadcom [source] and the processor in the NanoMind OBC in Suomi100 is a 32 MHz processor with

a single core [48]. While the data from the payload can be read and stored within the 31.25 microseconds with the RaspberryPi 3 to have a sample rate of 32 kHz at lowest, the reading and storing with NanoMind takes considerably longer, over 60 microseconds. In addition, we need to read the data two times before obtaining a completely new measurement value. This because the payload outputs the data from two audio channels, left and right, and we are reading the data from one channel at a time. Therefore, with the NanoMind OBC, we can theoretically have a sample rate of 8 MHz.

In addition, it was found that the reading and storing even at this rate is not consistent as the FreeRTOS runs other tasks while we are reading data from the payload. Figure 30 shows a screenshot from a *Tektronix TBS 1072B-EDU* oscilloscope reading the *slave select* pin from the payload. When the value of the slave select is low, a single reading of data from the instrument has happened. In an attempt to

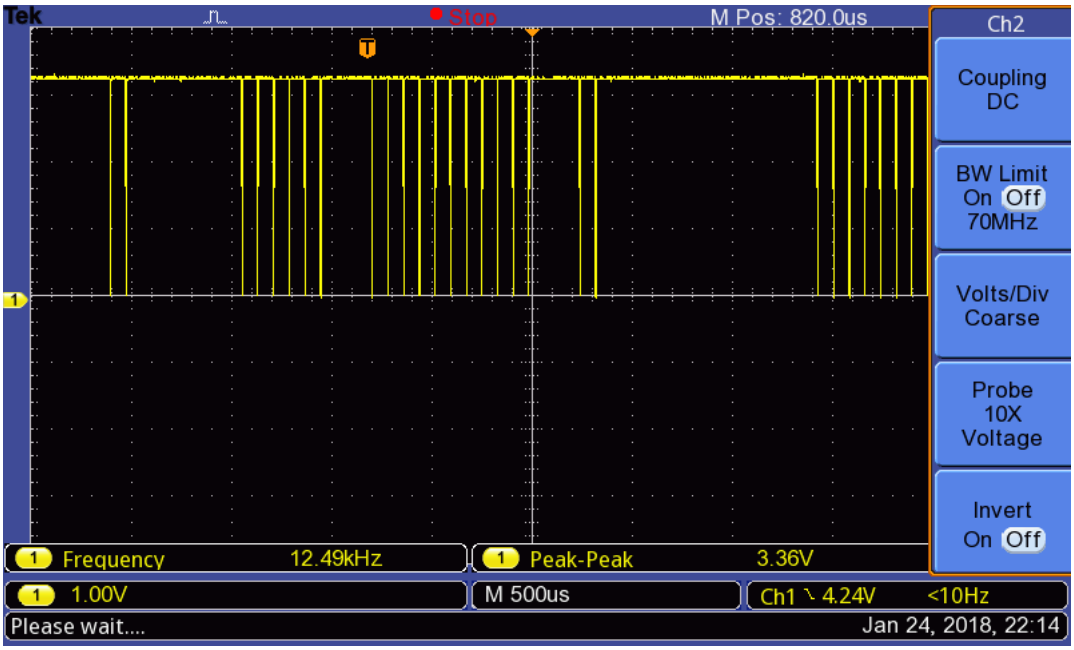

Figure 30: Screenshot from a Tektronix oscilloscope showing inconsistency in data reading rate from the radio payload instrument.

counter this, we first increased the priority of the FreeRTOS task assiocated with the operations of the radio instrument to highest among all the tasks in NanoMind. This didn't improve the situation though. Thus, we made our task to call for a FreeRTOS *vTaskSuspendAll()* command, which would give all the processing power of the processor for our task and freeze all the other tasks in the OBC. This caused some *watchdog* process unknown to us to reboot the computer, which possibly assumed that the computer was "frozen" and as a safety feature rebooted the computer.

Therefore, as a compromise the sample rate was dropped down to 1 kHz. This was done by adding a *vTaskDelay(1)* command to our FreeRTOS task after every read and store cycle. This was the shortest time we could add to our task, a shorter wait time would have possibly given a higher sample rate. But, this would have

required us to modify the general definitions of the NanoMind code, which could have caused unforeseen consequences to the operation of the satellite. Figure 31 shows that on certain periods we can obtain the data more consistently. Yet, during a longer time period, there still exists longer gaps between data reading events than what we could anticipate. Nonetheless, this data rate seems to be good enough for our needs. Though, any "real-time" radio noise that could be listened from the radio instrument by human ears is not feasible with this setup.

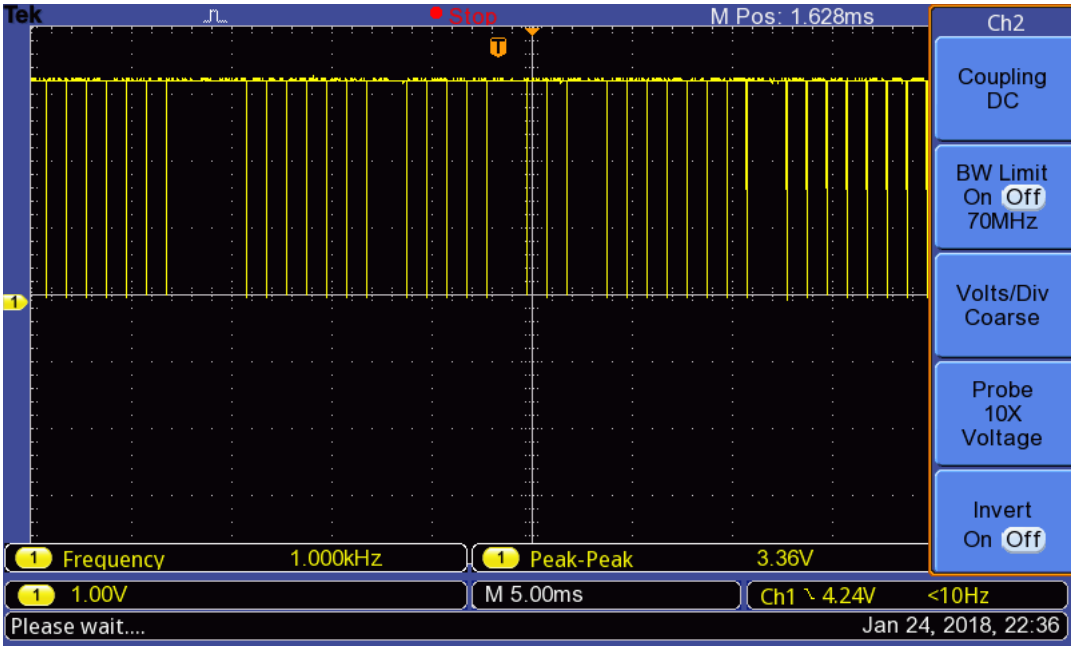

Figure 31: Screenshot from the oscilloscope showing better consistency with lower sample rate in data reading rate from the radio payload instrument.

In addition to the data rate problem, some problems were discovered with the payload and the Nanomind themselves. For the command to tune the frequency in the si4740 IC, an automatic antenna capacitance calculation was supposed to happen for a given frequency. Yet, from the Robot Framework log files it was found that the antenna capacitance value was always 1. Thus, we had write a separate command ourselves which calculates the capacitance value. In the consequent tests with this modification, the values seemed to be correct and it could also be seen from the figures plotted from the measurement data.

Another issue which was faced involved the GPIO pins in the NanoMind. We would have required four of them for our needs, yet only three out of six worked for us. By changing the purpose of these three pins in relation to our payload instrument was sufficient. This made it possible for us to read the data in the first place.

As a conlusion to the radio payload, the subsystem would have required its own processor and its own flash memory in order to have higher sample rate and with more consistent data reading and storing. As is the case with the NanoCam subsystem, which has a 536 MHz processor and 2 Gigabytes of non-volatile memory for image storage. Nonetheless, even with a very low sample rate, we can gather some valuable

information from the ionosphere [sources?].

4.1.3 Tests for basic satellite operations

Testing of the basic functionalities of the satellite software is necessary in order to obtain a reliable satellite. Therefore, tests were performed for satellite features such as safe satellite reboots, different housekeeping commands, flight planner commands and flight software updating. All of these can be seen as the basic functionalities which all of the other operations of the satellite depend upon. Test suites for these features don't follow the combinatorial testing methods used for two payload subsystems. Instead, test cases are based on different use cases or different situations we might end up facing with the satellite. In addition, the keywords used were less subsystem specific and more of the test cases used the generic keywords, such as *Send Command* and *Verify Reply Contained*.

Reboot tests

Twelve test cases were written for satellite reboots during different situations. The goal of these tests was to find out if during some operation, e.g. file upload, we can get the NanoMind "frozen" so that it doesn't reboot safely anymore. This test suite also has test cases simply to shutdown different subsystems and to verify their absence in the satellite bus with the simple *ping* command. From these test cases, it was found that the NanoCom communication system couldn't be shutdown completely. It still replied to ping commands even though a command was sent to shut it down. This functionality naturally is preferable (NanoCom manuaali sanoo tästä jotain?) and though this led to one test case of the reboot test suite to fail, the test can be seen to fail positively. As such, eleven test cases passed and one failed positively from this test suite.

The other types of test cases in the reboot test suite tested reboots during satellite operations and the reboots were mostly caused by adding reboot commands to the flight planner so that they would occur during some satellite operation, e.g. during radio payload operation. In all of these, the satellite came back safely. Yet, it has been several times that the NanoMind can get stuck and in those situations to get the OBC working again, it has required us to either manually reboot the EPS or to send a command via another subsystem to reboot the NanoMind. The satellite has recovered safely from these situations after reboot. But when we are using the satellite control software, CSP client, we are not able to do this. This situation arose the question whether there can happen something during orbit that causes the satellite to be "frozen" forever.

Fortunately, there are several watchdogs in the satellite that in theory can trigger a reboot after a certain time. Unfortunately, we were not able to deliberately trigger a situation where NanoMind is stuck and thus tests for these watchdog functionalities had to be omitted from the test suite.

Housekeeping tests

Another eleven test cases were performed to test the housekeeping features provided by the GomSpace platform. Test cases were written for different housekeeping commands of different subsystems as well as for housekeeping data storing and

transfer from the satellite. Testing of the beacon functionality was as well included in this test suite, as the beacon outputs recent HK information. Plotting of the beacon data was developed during the realization of these tests by another member of the satellite team, M.Sc Petri Koskimaa, and by the thesis author. These plotting functionalities were later used in the "day in the life of the satellite"-tests as well. All test cases of this test suite passed without errors. This means that, all the commands did what they were supposed to do, which was obviously the preferred result. In Figure N we have an example picture of a plot of the system current provided by the beacon at different timestamps.

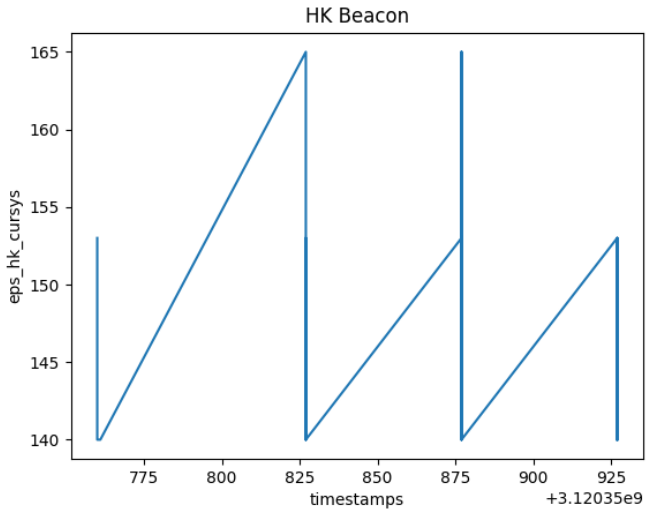

Figure 32: System current from the beacon data during imaging mode operation mode.

Flight planner tests

For testing of the flight planner feature, seven test cases were performed. The feature was tested with some basic flight planner creation commands as well as with more complicated ones. Out of all these tests involving basic functionalities of our satellite, this one had the most failed test cases. For first, it was assumed that giving the commands in wrong format (string instead of an integer) would cause the CSP client to indicate an error. Such a thing didn't happen but giving the commands in wrong format didn't cause the software to crash either. In addition, if the command string was too long, an error was indicated and no flight planner command was appended to flight plan list. Which happened to be the case when we tried to give one specific command for our radio payload, but this command to run the radio payload in one of the defined operation modes happens to be the single most essential command for the payload. Therefore, in order to make it work with the flight planner we need to modify the source code for the flight planner so that it can accept longer strings as commands (ei tehty vielä!). Otherwise, we can only use the radio payload when the satellite is in the reach of our communication radios. This is not preferrable as we would be radically limiting the area what we can measure.

Software update tests

Three test cases were performed for the software update feature. These test cases took a longer time to run than the other test cases involving the basic functionalities of the satellite, because we had to upload the new software to the satellite via the USB on the NanoUtil. These tests tested basic uploading of a new software image, rebooting back to the software that was flashed to NanoMind and tested uploading of an invalid file as an image to the new software. The satellite passed all these tests as expected giving us some confirmation that we actually can upload a new software to the satellite and command it to reboot with that software.

In the test case where we uploaded and invalid file as an image, the file itself was just a binary file containing measurement values from the payload radio. Booting with this file caused the NanoMind to crash with EXCEPTION 13 error message. In addition, we had set the NanoMind to try to boot with this file three times and it crashed each time with this error. Eventually as the boot counter reached zero, the satellite managed to recover the proper software it was flashed with. This test gave us knowledge that the satellite manages to recover itself if we happen to upload a software to the satellite that causes unexpected reboots.

4.1.4 Tests for "day in the life" operational scenarios of the satellite

Test suites were formed to test different phases of the scenarios that the satellite will encounter while in orbit. A scenario would be e.g. where we come from the eclipse, send commands to the satellite and downlink data. Each step in the scenario, like downlinking of data, formed its own test case within the test suite. Four different operational scenarios were tested. These are described as:

1: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered, the satellite takes an image and a measurement is made with the radio payload. Afterwards, the housekeeping data and the measurement are downloaded from the satellite. Finally, the satellite goes out of the reach of the groudstation.

2: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and flight planner is used to set the camera to take a picture while the satellite is in eclipse and out of the reach of the groundstation. After the satellite has orbited the Earth, the satellite comes again from the eclipse and is in the reach of the groundstation. Housekeeping data and the taken image is downloaded from the satellite. The satellite goes again out from the sight of the groundstation.

3: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and flight planner is used to set the camera to take pictures continuosly. Flight planner is used to gather housekeeping data continuosly as well. A reboot happens presence of all subsystems is verfified. The camera is given a command to take a picture to verify the basic operation of the

subsystem. In addition, verification for the charging of the batteries via the solar panels is verified. Finally, the housekeeping data is downloaded from the satellite.

4: The satellite comes from eclipse and is in the reach to form a radio link with the groundstation. Housekeeping data is gathered and flight planner is used to set the camera to take pictures continuosly. A file is uploaded to the satellite and finally the satellite goes out of sight of the groundstation.

One test suite for each scenario was written. Different phases of a scenario were made in to different test cases in the suite. Majority of the keywords used in the test suites were the *Persistent Command* and *Verify Reply Contained* keywords. Persistent commanding was required for majority of the commands because the radio link was not entirely stable. The test suites were executed several times and occasionally some keywords caused the test cases to fail due to the connection being temporarily lost.

Control of the solar simulator was not automated due it being just a lamp that one attaches to the electric plug to turn it on. Thus in practice the person responsible for the testing had to manually plug and unplug the simulator from the mains current. Therefore, another keywords were written for these tests which with a sound effect indicated that the lamp had to be turned or to be turned off. These were the *Wait And Notify*, *Notify After*, *Wait Until Time event* keywords.

One of the important aspect with these tests was verification that the solar panels charge the batteries. All the test suites had at least one test case for battery charging verification. In all such cases, the solar panels were detected and they did infact charge the batteries. As presented in section 2.1, one of the potential causes for failures with previous CubSats has been that the solar panels were not properly connected to the power bus [3]. Thus, testing of this was felt to be necessary. Figure 33 shows how the charge in the batteries changed over the execution of the first scenario. All the test cases for scenario 1 were executed succesfully. The housekeeping data and radio measurement were downloaded from the satellite succesfully via the radio link. Similarily, all test cases for scenario 3 succeeded. A reboot was caused in the satellite and all the subsystems responded to *ping* commands after the reboot. We were able to again take an image and verify that the solar panels were again charging the satellite. Test cases for scenario 4 were passed as well and a file was uploaded succesfully to the satellite.

Some problems emerged with testing of scenario 2. Namely, the download speed was too slow to enable us to download an image from the satellite during the time defined for the scenario. As presented in section 3.3.4, the time that the satellite is in the sight of the groundstation is approximately 10 minutes. The image taken by the camera was roughly 300 kilobytes in size and during the time we downloaded it, we received roughly 53 kilobytes.

One reason for the slow download speed was the setting for *rdpopt* command, which defines the wait times between each packets received among other things. With a short wait time the time between received packets is shorter, but the connection during downloading can be lost more frequently. With a longer wait time, the speed is lower but the connection is more stable. The theoretical maximum speed for

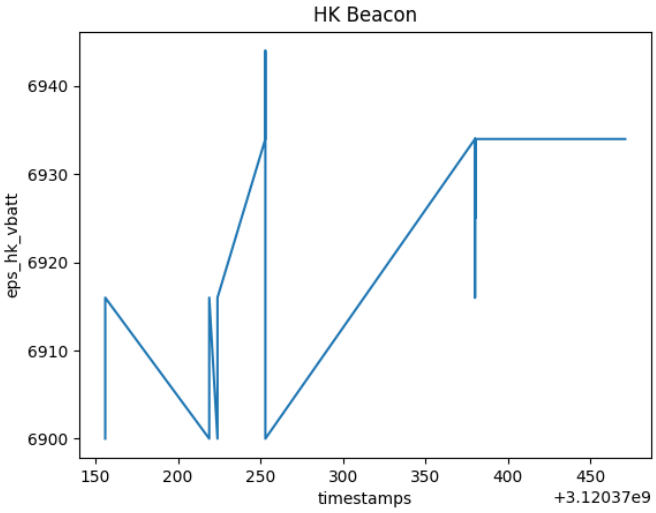

Figure 33: EPS battery charge from the beacon data during first defined satellite operational scenario.

downloading in our case is $0.9kb/s$. From the test suite logs we could see that at best we could receive data at the speed of $0.3kb/s$. Therefore, some fine tuning of the *rdpopt* command parameters would be needed to practically make the download faster.

As a conclusion, it was verified that the solar panels did charge the batteries and the communication with the satellite over the radio link worked. All the commands sent to the satellite worked as they were supposed to. Only the download speed was too slow so that we couldn't download a full picture from the satellite. Besides this, all the test cases passed.

4.2 Release version of CubeSatAutomation test library

The release version of the core function library, *CubeSatAutomation*, was cleared from Suomi100 related dependencies and only the generic process communication keywords are present. Explain some strategy in process communication (only some *Run Command* and *Run Command Persistently* keywords are used and the software knows how to choose between, send, write and type.)

Automation of several Linux terminal programs were tested with the test library succesfully. It was decided that library would be available for everyone who wishes to use test automation with terminal based programs (like some groundstation softwares), thus the library can be found in GitHub. In addition, some example Robot Framework test suites using the CubeSatAutomation library were added to the repository. Any team willing to use the library for test automation with Robot Framework would easily be able to start automating their groundstation software.

4.3 Developing operational scenario tests as requirement for CubeSat mission readiness: "Day in the Life of a CubeSat"

Currently the CubeSat standard demands the following tests to be performed for a satellite: *Random Vibration*, *Thermal Vacuum Bakeout*, *Shock Testing* and *Visual Inspection* [7]. These are demanded only for the reason of ensuring safe integration of the P-POD deployer and the CubeSat in to the launch vehicle. The specifications for these tests in fact usually are defined by the launch provider [7].

As can be seen, no testing is required for electrical or functional/operational testing of the satellite. In the research data represented in section 2.1 it was found that failure rates from 40 % to 20 % were prevalent in CubeSat missions [3, 17, 19]. In addition, it was suggested that these high failure rates were attributed to poor or nonexistent functional system integration testing. More so, understanding of integration and testing could have been something lacking from the University led CubeSat teams. In comparison, the CubeSat missions that were led by organisations and companies with vast experience in satellite integration and testing had considerably lower failure rates.

Therefore, we are suggesting at least for some form of guidelines for functional integration testing to be added to the CubeSat standard. Study for creation of such guidelines is deemed to be necessary. These guidelines could be devised from the experiences of previous succesful CubeSat missions as well as from the experiences and used testing practices of the "traditional" space missions.

In section 2.5, the represented research on failures with larger spacrafts also pointed to the lack of proper integration and testing as a source for mission failures. In addition, when the established testing practices used in NASA were "streamlined", consequent missions showed significant decrease in performance. When comparing the data from failed CubeSat and traditional space missions, it could further be claimed that at least some guidelines for integration testing are needed for the CubeSat standard.

In addition, a "Day in the life" operational scenario tests are required for NASA and ESA missions [33]. Besides the mechanical tests mentioned, a test following that principle could be devised for another test that is required from CubeSats. A test known as ***"Day in the life of a CubeSat***. Test like this would test the functionality and proper integration of the satellite. The communication with the groundstation could likewise be verified. The scenarios that would be tested could possibly be planned according to the guidelines given for functional testing of CubeSat. In theory, this test could decrease the amount of DOA cases for CubeSat missions. As noted in [3, 20] one of the alleged reasons for CubeSat failures has been improper integration of the solar panels to the satellite and thus not having enough power to form the radio link with the groundstation. A test like the one discussed here could on ground verify these two aspects of the mission.

One technical solution for a "Day in the life of a CubeSat" test is presented in this thesis. Which includes the integrated satellite, a solar simulator and the groundstation. Some basic scenarios for satellite operations were devised and tested.

The mechanical stress tests for CubeSats are performed with automated machinery, likewise a method to automate "Day in the life of a CubeSat" is presented in this thesis. This test automation includes Robot Framework and the function libraries developed by us. The final version of the function library was made to be a generic testing library, which is able to automate the use of many programs running in terminal environment. Therefore, many other groundstation control softwares could possibly be automated, given that they are terminal based. A setup for a "Day in the life of a CubeSat" test is presented below.

Documentation for a "Day in the life of a CubeSat" is another aspect that should be considered. Perhaps, the rigorous methods used in NASA [source] could be avoided and a more subtle method could be devised. Starting from design for operational scenarios to features to be tested and finally to documentation of passed and failed test cases.

Similar to testing for mission readiness at this level done by NASA and others, but "simpler"?

What kind of documentation? Is it affordable for all CubeSat teams?

4.4 Further lessons learned from testing of Suomi100 CubeSat

More rigorous requirement specifications, they would help higher level functional testing. What it should do, what it shouldn't do. Some area of operation, etc. More rigorous identification of operational scenarios and operation modes. All these in the vein of the CubeSat concept, not too rigorous or too formal, yet covering all the situations. Some examples from industry or other CubeSat missions?

5 Conclusions

In this thesis automated functional system integration tests for Suomi100 CubeSat were performed with Robot Framework. The need for testing at this level was identified from surveys conducted for all CubeSat missions flown previously, which showed a high failure rate for University led missions in contrast to low failure rates for missions performed by organisations and companies with established practices in integration and testing. Thus, testing methods used by e.g. NASA at satellite integration level along with industry proven testing practices were applied in design of testing performed for Suomi100. The testing was automated with Robot Framework, which is an industry proven acceptance testing framework. It was felt that doing the testing with the help of an automated computer software would give the testing more rigour and reproducibility. For automating the control of the satellite in interest of test automation, libraries for use with Robot Framework were programmed with Python programming language.

First tests conducted for the satellite were functional tests for integrated Suomi100 payloads, which are an optical camera and a radio instrument for ionospheric measurements. Second test set included acceptance testing of the core satellite functions such as housekeeping data collection, safe reboot handling, software updates and so forth. Final set consisted of tests for operational mission scenarios or "day in the life of the satellite"-tests.

An attempt was made with our limited resources to simulate the functional environment for testing of the payloads as well as for the operational scenario tests. For testing of the camera payload, the satellite was taken to a balcony at our department on a sunny day. During the testing of the integrated radio payload, a HackRF software defined radio was used to simulate the noise signals found in the ionosphere. A larger setup was used for the "day in the life of the satellite"-tests. With this test, we tested not just the proper functionality of the satellite software but that the radio link to the satellite works and that the solar panels can charge the satellite. Thus, the automated tests were performed via the radio link and a large Xenon lamp was used to simulate the Sun.

With the performed tests, we proved proper functionality of the camera payload as well as proper functionality for almost all of the tests conducted for the core satellite functions. The operational scenario tests showed that we can communicate with and send commands to the satellite via radio link and that the solar panels can charge the batteries in Suomi100. Only downside seemed to be the low downlink speed, yet this was to be expected.

The tests for the radio payload on the other hand presented several defects and most importantly emergent problems in the integration with the rest of the satellite platform. The defects were fixed with consequent software updates, but certain problems with integration couldn't be overcome. The biggest one being the slow speed of reading and storing of data from the payload by the OBC. This forced us to drop the sample rate from the possible 32-48 kHz to just 1 kHz for reliable data measurement.

Performing of these tests proved to us that Robot Framework can work as a

testing framework for CubeSats and that we could carry out automated testing of Suomi100 CubeSat. The software libraries doing the actual automation were developed into a generic testing library, which potentially could be used for testing and automation of any Linux terminal software, such as a satellite ground station control software.

– Did we improve the functionality and reliability of the satellite?

References

[1] Collins, Martin *After Sputnik: 50 Years of the Space Age*, Smithsonian Books, HarperCollins, New York, USA, 2007.

[2] Belfore, Michael, *Rocketeers: How a Visionary Band of Business Leaders, Engineers, and Pilots Is Boldly Privatizing Space*, 1st edition, Smithsonian Books, HarperCollins, New York, USA, 2007.

[3] Swartwout, Michael, *The First One Hundred CubeSats: A Statistical Look*, Journal of small satellites, 2013.

[4] *http://pluto.jhuapl.edu/*, obtained 8th of March 2018.

[5] Pratt, Timothy, Charles W. Bostian & Jeremy E. Allnutt, *Satellite Communications*, 2nd edition, John Wiley & Sons, New Jersey, USA, 2003.

[6] *https://tem.fi/en/spacelaw*, obtained 23rd of February 2018.

[7] The CubeSat Program, *CubeSat Design Specification Rev. 13*, California Polytechnic State University, 2014.

[8] *http://markets.businessinsider.com/news/stocks/iceye-successfully-launches-world-s-first-sar-microsatellite-and-establishes-finland-s-first-commercial-satellite-operations-1012996541*, obtained 23rd of February 2018.

[9] *https://www.nasa.gov/mission_pages/station/research/benefits/cubesat*, obtained 23rd of February 2018.

[10] Fortescue, Peter et al., *Spacecraft Systems Engineering*, 4th edition, John Wiley & Sons, USA, 2011.

[11] Straub, Jeremy et al., *OpenOrbiter: A Low-Cost, Educational Prototype CubeSat Mission Architecture*, Machines 1:1-32, 2013.

[12] Bouwmeester Jasper et al, *Survey on the implementation and reliability of CubeSat electrical bus interfaces*, CEAS Space J, 9:163-173, 2017.

[13] Laplante, Phillip A. & Ovaska Seppo J., *Real-Time Systems Design and Analysis*, 4th edition, IEEE Press, USA 2012.

[14] Myers, Glenford J., *The Art of Software Testing*, 3rd edition, John Wiley & Sons, New Jersey, USA, 2012.

[15] Pries, Kim H. & Quigley Jon M., *Testing Complex and Embedded Systems*, Taylor and Francis group, CRC Press, USA 2011.

[16] Kandl, Susanne et al., *A Formal Approach to System Integration Testing*, European Dependable Computing Conference, United Kingdom, 2014.

[17] Swartwout, Michael, *Secondary Spacecraft in 2016: Why Some Succeed (And Too Many Do Not)*, Saint Louis University , 2016 IEEE Aerospace Conference, Montana, USA, 5-12 March 2016.

[18] Doncaster, Bill & Williams Caleb & Shulman Jordan, *2017 Nano/Microsatellite Market Forecast*, SpaceWorks Enterprises, Inc. (SEI), Atlanta, USA, 2017.

[19] Swartwout, Michael, *Secondary Spacecraft in 2015: Analyzing Success and Failure*, Saint Louis University, 2015 IEEE Aerospace Conference, Montana, USA, 7-14 March 2015.

[20] Langer, Martin & Bouwmeester Jasper, *Reliability of CubeSats - Statistical Data, Developer's Beliefs and the Way Forward*, 30$^{\text{th}}$ Annual AIAA/USU Conference on Small Satellites, 2016.

[21] *http://spacenews.com/cubesat-reliability-a-growing-issue-as-industry-matures/*, obtained 12th of December, 2017.

[22] *https://www.nasa.gov/smallsat-institute*, obtained 12th of December, 2017.

[23] *https://www.space.com/13558-historic-mars-missions.html*, obtained 23rd of February 2018.

[24] Tolker-Nielsen, Toni, *EXOMARS 2016 - Schiaparelli Anomaly Inquiry*, European Space Agency, 2017.

[25] Tafazoli, Mak, *A study of on-orbit spacecraft failures*, Acta Astronautica, 64:195-205, 2009.

[26] Castet, Jean-Francois et al., *Satellite and satellite subsystems reliability: Statistical data analysis and modeling*, Georgia Institute of Technology, Reliability Engineering and System Safety, Vol 94, 2009.

[27] Tosney, William F. et al., *Satellite verification planning: Best practices and pitfalls related to testing*, Proceedings of the 5$^{\text{th}}$ International Symposium on Environmental Testing for Space Programmes, Netherlands, 2004.

[28] Jackelen George. et al, *When Standards and Best Practices are Ignored*, Fourth IEEE International Symposium and Forum on Software Engineering Standards, 1999.

[29] *http://www.thespacereview.com/article/1579/1*, obtained 5th of March 2018.

[30] Tomayko, James E., *Computers in Spaceflight: The NASA experience*, National Aeronautics and Space Administration, Wichita State University, USA 1988.

[31] Wortman, Kristin, *Management of Independent Software Acceptance Test in the Space Domain: A Practitioner's View*, 2012 IEEE Aerospace Conference, Montana, USA, 3-10 March 2012.

[32] Badaruddin, Kareem S., *System Testbed Use on a Mature Deep Space Mission: Cassini*, 2008 IEEE Aerospace Conference, Montana, USA, 1-8 March 2008.

[33] White, Julia D., *Test Like You Fly: Assessment and Implementation Process*, Aerospace Report NO. TOR-2010(8591)-6, Space and Missile Systems Center Air Force Space Command, El Segundo California, 2010.

[34] European Cooperation For Space Standardization (ECSS) Secretariat, *Space Engineering Verification*, ESA Requirements and Standards Division, ESTEC, Netherlands, 2009.

[35] Williams, W. C., *Lessons from NASA: Design reviews, failure analysis, and redundancy are favored over predictions and life testing for spacecraft success*, IEEE Spectrum, Vol. 18:10, 1981.

[36] Jacklin, Stephen A., *Survey of Verification and Validation Techniques for Small Satellite Software Development*, Space Tech Expo Conference USA, CA, 2015.

[37] , Tatsumi, Keizo, *Test Automation - Past, Present and Future*, System Test Automation Conference 2013, Tokyo, Japan, 1st of December 2013.

[38] Subramanyan, Rajesh et al., *10^{th} International Workshop on Automation of Software Test (AST 2015)*, 37^{th} IEEE International Conference on Software Engineering, Firenze Italy, 16 May - 24 May 2015.

[39] Kasurinen, Jussi et al., *Software Test Automation in Practice: Empirical Observations*, Lappeenranta University of Technology, Advances in Software Engineering, Vol. 2010, Hindawi Publishing Corporation, 2010.

[40] Cervantes, Alex, *Exploring the Use of a Test Automation Framework*, Jet Propulsion Laboratory, 2009 IEEE Aerospace Conference, Montana, USA, 7-14 March 2009.

[41] Pajunen, Tuomas et al., *Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework*, Tampere University of Technology, 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21-25 March 2011.

[42] *http://robotframework.org/*, obtained 28th of January 2018.

[43] Laukkanen, Pekka, *Data-Driven and Keyword-Driven Test Automation Frameworks*, Helsinki University of Technology, 2006.

[44] *http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html*, obtained 6th of March 2018.

[45] *http://www.aalto.fi/fi/current/news/2015-06-24/*, obtained 6th of March 2018.

[46] *http://www.suomi100satelliitti.fi/mika*, obtained 6th of March 2018.

[47] *https://gomspace.com/example-configurations.aspx*, obtained 6th of March 2018.

[48] *NanoMind A3200 datasheet*, gs-ds-nanomind-a3200-1.10, GomSpace A/S, 2016.

[49] *NanoPower P31 Datasheet*, gs-ds-nanopower-p31u-17, GomSpace A/S, 2017.

[50] *NanoCom AX100 Datasheet*, gs-ds-nanocom-ax100-3.3.docx3.3, GomSpace A/S, 2016.

[51] *NanoCam C1U Datasheet*, gs-ds-nanocam-c1u-1.5, GomSpace A/S, 2017.

[52] *Si4740/41/42/43/44/45-C10 Automotive AM/FM Radio Receiver*, Silicon Laboratories, 2009.

[53] *Si47xx Programming Guide*, Rev. 1.0 9/14, Silicon Laboratories, 2014.

[54] Koskimaa, Petri, *Ferrite Rod Antenna in a Nanosatellite Medium and High Frequency Radio*, Aalto-University School of Electrical Engineering, 2016.

[55] *A3200 SDK*, gs-man-nanomind-a3200-sdk-v1.2, GomSpace A/S, 2017.

[56] *The FreeRTOS Reference Manual*, version 10.0.0 issue 1, Amazon Web Services, 2017.

[57] *AVR32 Tool Chain*, gs-man-avr32-toolchain-1.4, GomSpace A/S, 2016.

[58] Kerrisk, Michael *The Linux Programming Interface*, No Starch Press Inc., San Francisco, USA, 2010.

[59] Kallio, Esa et al., *Feasibility study for a nanosatellite-based instrument for in-situ measurements of radio noise*, 1st URSI Atlantic Radio Science Conference (URSI AT-RASC), Las Palmas, Spain, 16-24 May 2015.

A CubeSatAutomation function library

The source code for the release version of the CubeSatAutomation test automation library is presented in this section. The code can also be found in Github with an automatic installer. Required libraries are:?

```
import socket
import sys
import os
import signal
import binascii
import math
import subprocess
import thread
import time                 # For time stamps
import robot
from time import sleep
from fcntl import fcntl, F_GETFL, F_SETFL
from os import O_NONBLOCK, read
from ConfigParser import SafeConfigParser
import numpy as np
import matplotlib.pyplot as plt

class CubeSatAutomation:
        ''' Implement cleaned version and verify its operation with some linux
                Use some QWeb stuff.
        '''

        # Only one instance of the class for all test cases

        ROBOT_LIBRARY_SCOPE = 'TEST_SUITE'
        proc = None
        server = None
        port = 0
        sock = None
        writing = False
        writing_done = False
        reply_buffer = ""
        operation_timer = 0

        def __init__(self):
                self.parser = SafeConfigParser()

        def connect_socket(self, config_file=None, server=None, port=None):
                '''         ADD DOCUMENTATION
                '''
```

```
        if config_file:
                self.parser.read(str(config_file))
        if server is None:
                self.server = self.parser.get('SOCKET', 'server')
                print "Read:"
                print self.server
        else:
                self.server = str(server)
        if port is None:
                self.port = self.parser.get('SOCKET', 'port')
                print "Read:"
                print self.port
        else:
                self.port = str(port)
        #self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        CubeSatAutomation.sock = socket.socket(socket.AF_INET, socket.SOC
        server_address = (str(self.server), int(self.port))
        print 'connecting to %s port %s' % server_address
        CubeSatAutomation.sock.connect(server_address)
        print CubeSatAutomation.sock

# Kaikki None aikasemmin ja esim if config_file is None:'
#                 self.parser.read('s100_config.cfg')
#         else:
#                 self.parser.read(str(config_file))'
def client_start(self, config_file=None, prog=None, params=None):
        ''' ADD DOCUMENTATION
        '''

        if "None" in str(config_file):
                print "None in config file"
                pass
        else:
                self.parser.read(str(config_file))
        if prog is None:
                prog = self.parser.get('PROGRAM', 'path')
                print "Read:"
                print prog
        elif "None" in str(prog):
                prog = ""
        else:
                print "Prog" is str(prog)
                prog = str(prog)
        if params is None:
```

```
                params = self.parser.get('PROGRAM', 'params')
                print "Read:"
                print params
        elif "None" in str(params):
                 params = ""
        else:
                params = str(params)
        CubeSatAutomation.proc = subprocess.Popen([str(prog) + " " + str(
                stdout=subprocess.PIPE, stderr=subprocess.STDOUT, shell=T
        flags = fcntl(CubeSatAutomation.proc.stdout, F_GETFL) # get curre
        print "fcntl flags:"
        print flags
        fcntl(CubeSatAutomation.proc.stdout, F_SETFL, flags | O_NONBLOCK)
        print "Started process " + str(self.proc) + " with parameters " +

def client_close(self, prog, socket_comm=None):
        if socket_comm:
                CubeSatAutomation.sock.shutdown(socket.SHUT_RDWR)
                CubeSatAutomation.sock.close()                          #Cu
        if prog:
                                        subprocess.Popen([str(kill_comman
        else:
                raise OSError ("Program to be closed must be defined!")

def monitor_start(self):
        ''' Start a terminal window and send the keywords, commands and
        '''
        pass

def send_message(self, message):
        print "Sending message %s to socket" % message
        message = message + "\r"
        try:
                CubeSatAutomation.sock.sendall(message)
        except:
                print "Failed to send message to socket, trying to write
                try:
                        CubeSatAutomation.proc.stdin.write(command)
                except:
                        print "Failed to write message to stdin, trying t
                        import pyautogui
                        pyautogui.typewrite(str(message))
                        pyautogui.press('enter')
        sleep(1)
```

```
def send_command(self, message, option="Store", timeout=5, read_timeout=5
        self.send_message(str(message))
        console_lines = self.read_console_reply(int(timeout), int(read_ti
        console_lines = str(console_lines).split("\\n")
        if "Store" in str(option):
                CubeSatAutomation.reply_buffer = console_lines
                print "Console lines"
                print CubeSatAutomation.reply_buffer
        for line in console_lines:
                if "error -1" in line or "error -2" in line:
                        raise ValueError("%s" % line)


def write_command(self, message, option="Store", timeout=2, read_timeout=
        command = str(message) + '\r'
        CubeSatAutomation.proc.stdin.write(command)
        console_lines = self.read_console_reply(int(timeout), int(read_ti
        console_lines = str(console_lines).split("\\n")
        if "Store" in str(option):
                CubeSatAutomation.reply_buffer = console_lines
                print "Console lines"
                print CubeSatAutomation.reply_buffer

def type_command(self, message, option="Store", timeout=2, read_timeout=2
        '''         pyautogui is needed. Some solution without it?
        '''
        import pyautogui
        pyautogui.typewrite(str(message))
        pyautogui.press('enter')
        console_lines = self.read_console_reply(int(timeout), int(read_ti
        console_lines = str(console_lines).split("\\n")
        if "Store" in str(option):
                CubeSatAutomation.reply_buffer = console_lines
                print "Console lines"
                print CubeSatAutomation.reply_buffer

def read_console_reply(self, timeout, read_timeout=10):
        console_lines = []
        time_count = 0
        while time_count < int(timeout):
                sleep(1)        # Wait for data to be 'cooked'
                time_count = time_count + 1
                try:
                        line = read(CubeSatAutomation.proc.stdout.fileno(
                except OSError:         # No data to be read, wait if mor
```

```
                                console_lines.append("Waiting for more data from
                                read_timecount = 0
                                while read_timecount < int(read_timeout):
                                        try:
                                                line = read(CubeSatAutomation.pro
                                        except OSError:
                                                sleep(1)
                                                read_timecount = read_timecount +
                                                continue
                                        else:
                                                break
                                if read_timecount >= int(read_timeout):
                                        console_lines.append("Process data read t
                                        break
                        print "term:" + line.rstrip()
                        if 'exit_client' in line:
                                os.killpg(os.getpgid(CubeSatAutomation.proc.pid),
                                break
                        else:
                                if line != '':
                                        console_lines.append(line)
                console_lines_str = console_lines
                console_lines = ''.join(console_lines_str)
                console_lines = str(console_lines).split("\\n")         # Why does
                print console_lines
                return console_lines

        def clear_replies(self, option="None", read_timeout=5):
                ''' Clear process replies with flush command
                '''
                CubeSatAutomation.proc.stdout.flush()
                if "Stored" in str(option) or "All" in str(option):
                        CubeSatAutomation.reply_buffer = ""
                if "Stored" not in str(option):
                        try:
                                read(CubeSatAutomation.proc.stdout.fileno(), 1024
                        except OSError:         # No data to be read, wait if mor
                                read_timecount = 0
                                while read_timecount < int(read_timeout):
                                        try:
                                                read(CubeSatAutomation.proc.stdou
                                        except OSError:
                                                sleep(1)
                                                read_timecount = read_timecount +
                                                continue
```

```
                                else:
                                        break


        def store_client_responses_thread(self, filename, timeout, read_timeout):
                """         Write the responses to a file.
                            Thus we can keep track of what we have done with the
                            Various verify_result keywords parse the responses a
                            Use the Queue module to send information to thread(s
                            Or some threading. Queue as a class variable? Or jus
                            Close this if after few loops we get nothing.
                 """
                #CubeSatAutomation.read_queue = Queue.Queue()
                time_count = 0
                while time_count < int(timeout):
                        if CubeSatAutomation.writing is False:
                                try:
                                        f = open(str(filename), 'a')        # Cre
                                except IOError:
                                        print "Storing couldn't open %s" % str(fi
                                        raise IOError ("Couldn't open %s" % str(
                                console_lines = []
                                console_lines = self.read_console_reply(10, int(r
                                time_count = time_count + int(read_timeout)
                                f.writelines(console_lines)
                                f.close()
                                CubeSatAutomation.writing = True        # Lock th
                        else:
                                sleep(1)
                                time_count = time_count + 1
                        if CubeSatAutomation.writing_done is True:
                                CubeSatAutomation.writing = False
                                CubeSatAutomation.writing_done = False
                                break
                # SVA style solution, write some end of test indicator to the f
                # Add time to the filename
                        #sys.exit()
                # Write beginning time and end time to the file

        def store_client_responses(self, filename, timeout=5, read_timeout=20):
                thread.start_new_thread(self.store_client_responses_thread, (file

        def verify_reply_message(self, message, timeout=5, read_timeout=10):
                ''' Read messages from the process stdout and verify if the des
                 '''
```

```
        console_lines = self.read_console_reply(int(timeout), int(read_ti
        console_lines = str(console_lines).split("\\n")
        found = False
        for line in console_lines:
                if str(message) in line:
                        found = True
                        break
        if not found:
                print console_lines
                raise ValueError ("Message %s was not found in the proces

def verify_reply_contains(self, message, timeout=5, read_timeout=10):
        self.verify_reply_message(message, timeout, read_timeout)

def verify_reply_contains_not(self, message, timeout=5, read_timeout=10):
        console_lines = self.read_console_reply(int(timeout), int(read_ti
        console_lines = str(console_lines).split("\\n")
        found = False
        for line in console_lines:
                if str(message) in line:
                        found = True
                        break
        if found:
                print console_lines
                raise ValueError ("Message %s was not supposed to be fou

def verify_reply_contained(self, message):
        console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
        found = False
        for line in console_lines:
                if str(message) in line:
                        found = True
                        break
        if not found:
                print console_lines
                raise ValueError ("Message %s was not found in the recent

def verify_reply_contained_not(self, message):
        console_lines = str(CubeSatAutomation.reply_buffer).split("\\n")
        found = False
        for line in console_lines:
                if str(message) in line:
                        found = True
                        break
        if found:
```

```
                print console_lines
                raise ValueError ("Message %s was not supposed to be fou

        def wait_until_reply_contains(self, message, timeout=20, read_timeout=5):
                completed = False
                found = False
                time_count = 0
                for line in CubeSatAutomation.reply_buffer:
                        if str(message) in str(line):
                                completed = True
                                found = True
                while not completed:
                        console_lines = self.read_console_reply(1, int(read_timeo
                        console_lines = str(console_lines).split("\\n")
                        time_count = time_count + 1
                        sleep(1)
                        if time_count > int(timeout):
                                completed = True
                        for line in console_lines:
                                if str(message) in line:
                                        found = True
                                        completed = True
                                        break
                if not found:
                        print console_lines
                        raise ValueError ("Message %s was not found in the proces

        def wait_until_reply_contains_not(self, message, timeout=20, read_timeout
                completed = False
                time_count = 0
                while not completed:
                        console_lines = self.read_console_reply(1, 10)
                        console_lines = str(console_lines).split("\\n")
                        time_count = time_count + 1
                        print time_count
                        sleep(1)
                        if time_count > int(timeout):
                                completed = True
                        print console_lines
                        completed = True
                        for line in console_lines:
                                if str(message) in line:
                                        print "found, not finished"
                                        completed = False
                                        break
```

```
        if time_count > int(timeout):
                print console_lines
                raise ValueError ("Message %s was found in the process re

    def verify_stored_reply_message(self, message, filename, timeout=30):
        completed = False
        time_count = 0
        while not completed:
                time_count = time_count + 1
                sleep(1)
                if time_count > int(timeout):
                        print "Time count is larger? " + str(time_count)
                        CubeSatAutomation.writing = False
                        CubeSatAutomation.writing_done = True
                        completed = True
                        break
                if CubeSatAutomation.writing is True:
                        try:
                                f = open(str(filename), 'r')
                        except IOError:
                                raise IOError ("Couldn't open %s" % str(
                        console_lines = f.readlines()
                        for line in console_lines:
                                if str(message) in line:
                                        f.close()
                                        CubeSatAutomation.writing = False
                                        CubeSatAutomation.writing_done =
                                        completed = True
                                        break
                        f.close()
                        CubeSatAutomation.writing = False        # Free
                        #CubeSatAutomation.writing_done = True
                else:
                        continue
        # Move the file storing to another keyword, but we want to store
        if completed:
                try:
                        f = open(str(filename), 'r')
                except IOError:
                        print "Verifying couldn't open %s\n" % str(filena
                console_lines = f.readlines()
        f.close()
        # Rename and move file
        new_filename = str(os.getcwd()) + "/satellite_passes/"  + str(fil
        os.rename(str(filename), new_filename)
```

```
        if time_count > int(timeout):
                raise ValueError("Message %s was not found in the stored


def wait_and_notify(self, event, timeout, soundfile):
        time_count = 0
        while True:
                if time_count >= int(timeout):
                        break
                time_count = time_count + 1
                time.sleep(1)
        # Play some sound signal when time is reached
        cwd = os.getcwd()
        path = str(cwd) + str(soundfile)
        from playsound import playsound
        playsound(path)

def notify_after_thread(self, event, timeout, soundfile):
        time_count = 0
        while True:
                if time_count >= int(timeout):
                        break
                time_count = time_count + 1
                CubeSatAutomation.operation_timer = CubeSatAutomation.ope
                time.sleep(1)
        # Play some sound signal when time is reached
        cwd = os.getcwd()
        path = str(cwd) + str(soundfile)
        from playsound import playsound
        playsound(path)

def notify_after(self, event, timeout, soundfile):
        thread.start_new_thread(self.notify_after_thread, (event, timeout

def wait_until_time_event(self, event, timeout):
        while True:
                if CubeSatAutomation.operation_timer >= int(timeout):
                        print "Timed event %s reached" % str(event)
                        CubeSatAutomation.operation_timer = 0
                        time.sleep(20)          # Average time for the no
                        break

def persistent_command(self, message, exception_replies, end_reply="None"
        ''' Sends a command persistently until either time runs out or
          '''
```

```
time_count = 0
completed = False
found = False
error_found = False
command = str(message) + '\r'
CubeSatAutomation.proc.stdin.write(command)
exception_replies = str(exception_replies)
exception_replies = exception_replies.split(';')
print exception_replies
while not completed:
        if time_count >= int(timeout):
                completed = True
                break
        console_lines = self.read_console_reply(10, int(read_time
        console_lines = str(console_lines).split("\\n")
        CubeSatAutomation.reply_buffer = console_lines
        for line in console_lines:
                for exception_reply in exception_replies:
                        print "exception_reply:" + str(exception_
                        print "reply line:" + str(line)
                        if str(exception_reply) in str(line):
                                print "Exception %s found, retryi
                                #error_found = True
                                CubeSatAutomation.proc.stdin.writ
                                break
                # if len(end_reply) > 1:
                if str(end_reply) in str(line):
                        completed = True
                        found = True
                        break
                if "None" in str(end_reply):
                        completed = True
                        found = True
                        break
        time_count = time_count + 1
        time.sleep(1)
if len(end_reply) > 1:
        if found:
                print "Desired reply %s was found in process repl
        else:
                if str(end_reply) == "Timeout":
                        pass
                else:
                        raise ValueError ("Desired reply %s was
if "None" in str(end_reply) or "Timeout" in str(end_reply):
```

```
console_lines = self.read_console_reply(1, int(read_timeo
console_lines = str(console_lines).split("\\n")
CubeSatAutomation.reply_buffer = console_lines
for line in console_lines:
                print "exception_reply:" + str(exception_
                print "reply line:" + str(line)
                if str(exception_reply) in str(line):
                        raise ValueError ("Exception %s s
```

B Robot Framework test suites

```
*** Settings ***
Library   String
Library   ../libraries/CubeSatAutomation.py
Library   ../libraries/RadioPayload.py
Library   ../libraries/NanoCam.py
Resource  ../resources/s100_keywords.robot
Suite Setup   Client Start   None    /home/petri/s100/EGSE/csp-client-v1.1/build/
Suite Teardown  Client Close   None


*** Test Cases ***

Come from eclipse - Verify charging
    [Documentation]                 Day in the life operations
    [Tags]                          OPMODE-POWER
    Wait and Notify                 Coming from eclipse     5   /resources/notify
    Notify After                    Going to eclipse        600     /resources/nc
    Satellite State                 Unknown
    Clear Replies                   All
    Persistent Command                  reboot 1   error
    Sleep                           10
    Persistent Command                   rdpopt 5 30000 16000 1 2000 3     error
    Persistent Command                 hk get 0 10 10 0 /flash/hk_robot.dat
    Persistent Command                   gssbcsp addr 6     error
    Persistent Command                   gssbcsp interstage sensors     error
    Verify Reply Contained Not      Coarse Sunsensor: 0     # These to 0 when we
    Persistent Command                   gssbcsp addr 7     error
    Persistent Command                   gssbcsp interstage sensors     error
    Verify Reply Contained Not      Coarse Sunsensor: 0     # These to 0 when we
    Persistent Command                  eps hk     error   Voltage
    Persistent Command                   eps hksub vi     error    Vbatt
    Verify Reply Contained          Vbatt
    Verify Reply Contained          Isun
    Verify Reply Contained          Isys
    Verify Reply Contained Not      boost[1] 0mV
    Verify Reply Contained Not      boost[2] 0mV
    Persistent Command                ftp server 1       error
    Run Keyword And Ignore error    Persistent Command    ftp rm /flash/hk_robot.
    Persistent Command                rparam download 1 19       error      Wrote
    Persistent Command               rparam set col_en 1         error      Resul
    Persistent Command               rparam set store_en 1           error  Resul
    Persistent Command               rparam send         error      REP
    Persistent Command                rparam download 1 18           error  Wrote
```

```
    Persistent Command              rparam set bcn_interval 10 10 10        erro
    Persistent Command              rparam send         error      REP
    Verify Reply Contained Not  error
    Persistent Command               hk get 0 10 10 0 /flash/hk_robot.dat

# Add ADCS commands

Come from eclipse - Take image
    [Documentation]                 Day in the life operations
    [Tags]                          OPMODE-IMAGING
    Satellite State                 Communicating
    Clear Replies                   All
    Persistent Command                   hk get 0 10 10 0 /flash/hk_robot.dat   e
    Persistent Command                   rparam download 1 19   error   Wrote
    Persistent Command                   rparam set col_en 0    error   Result
    Persistent Command                   rparam set store_en 0  error   Result
    Persistent Command                   rparam send            error   REP
    Persistent Command                 rparam download 1 18     error   Wrote
    Persistent Command                rparam set bcn_interval 0 0 0      error   R
    Persistent Command                rparam send               error       REP
    Persistent Command                   adcs server 1 20       error
    Persistent Command                   adcs ephem tle new     error
    Persistent Command                   adcs run start         error
    Persistent Command                   adcs set nadir         error
    Persistent Command                    cmp route_set 6 1000 8 1 CAN  error   S
    Persistent Command                    cam snap -a   Snap error  All
    Persistent Command                    cam store test.jpg    error    Result
    Persistent Command                    hk get 0 10 10 0 /flash/hk_robot.dat  e

Come from eclipse - Record radio signals
    [Documentation]                 Day in the life operations
    [Tags]                          OPMODE-LOWOBS
    Satellite State                 Communicating
    Clear Replies                   All
    Persistent Command                    hk get 0 10 10 0 /flash/hk_robot.dat  e
    Persistent Command                    adcs run fullstop     error
    Persistent Command                 rparam download 1 19     error   Wrote
    Persistent Command                rparam set col_en 1       error   Result
    Persistent Command                rparam set store_en 1     error   Result
    Persistent Command                rparam send               error       REP
    Persistent Command                 rparam download 1 18     error       Wrote
    Persistent Command                rparam set bcn_interval 10 10 10       error
    Persistent Command                rparam send                   error       R
    Persistent Command                 hk get 0 10 10 0 /flash/hk_robot.dat
    Write Command                     radio operation /flash/radio_params.cfg /fla
```

```
    Sleep                          120
    Persistent Command                  hk get 0 10 10 0 /flash/hk_robot.dat  e

Come from eclipse - Downlink data
    [Documentation]                Day in the life operations
    [Tags]                         OPMODE-COMM
    Satellite State                Communicating
    Clear Replies                  All
    Persistent Command                   rdpopt 6 30000 16000 1 2000 3     error
    Persistent Command                  rparam download 1 19    error      Wrote
    Persistent Command                  rparam set col_en 0     error      Resul
    Persistent Command                  rparam set store_en 0   error      Resul
    Persistent Command                  rparam send             error      REP
    Persistent Command                 rparam download 1 18       error      Wrote
    Persistent Command                 rparam set bcn_interval 0 0 0      error   R
    Persistent Command                 rparam send                error
    Persistent Command                      ftp server 1          error
    Persistent Command                 ftp download_file /flash/data/m2_debug.dat m2
    Persistent Command                      hk get 0 10 10 0 /flash/hk_robot.dat
    Persistent Command                      ftp server 1          error
    Persistent Command                 ftp download_file /flash/hk_robot.dat hk_robo
    Wait Until Time Event          Going to eclipse    600
    Parse HK                       hk_robot.dat  None    True    hk_plot1.png
    Parse HK                       hk_robot.dat  None    True    hk_plot2.png
    Log                            <img src="hk_plot1.png" width="500" height="5
```