Embedded Systems & Applications

TECHNISCHE UNIVERSITÄT DARMSTADT

Specification and Partial Implementation of a Test and Control Framework for the OPS-SAT Project

ops-sat

Bachelorthesis by

Moritz Woitelle

September 2017

Examiner:
Prof. Dr.-Ing. Andreas Koch

Supervisor:
Dr.-Ing. Andreas Engel

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Specification and Partial Implementation of a Test and Control Framework for the OPS-SAT Project

Spezifikation und teilweise Implementierung einer Test- und Steuerarchitektur für das OPS-SAT Projekt

Bachelorthesis by Moritz Woitelle
Submitted on 11.09.2017
Examiner: Prof. Dr.-Ing. Andreas Koch
Supervisor: Dr.-Ing. Andreas Engel

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, 11. September 2017

(Moritz Woitelle)

Acknowledgements

First and foremost, I would like to thank David Evans and Andreas Engel. As my supervisors from the European Space Agency and the Embedded Systems and Applications Group of the Department of Computer Science, they guided me throughout the process of writing this thesis.

I would also like to thank Manuel Kubicka and Claudiu Cherciu for always answering every question I had, and for all the support during the time at ESOC.

Abstract

When there is an innovation in satellite operations, it is very hard to test it. This is due to the risk associated with changing operational methods or software on an ongoing satellite mission. If there would be an error uploaded with the software, the operators would lose an otherwise healthy satellite.

Also, space missions are handicapped by the fact that the on-board computer (OBC) has to withstand the radiation environment of outer space. The terrestrial chip industry doubles the computing power of computer chips every year. To make computer chips radiation hardened takes time and is cost intensive, so the radiation hardened hardware cannot keep up with this doubling. This leads to more and more commercially available off the shelf (COTS) hardware being used to build satellites. But using COTS hardware has its downsides. Only by applying special techniques, such as redundancy, the COTS hardware can survive as long as radiation hardened hardware in space radiation environment. To further research the reliability of COTS components is a challenge.

The European Space Agency (ESA) tackles those two known challenges in spaceflight with its new satellite called Operations-Satellite (OPS-SAT). Its main objective is to be an orbital laboratory for experiments in the satellite operations sector. Any European institution or company can enlist for testing their experiments. The experiments can control the satellite, even if their creators do not have any experience in satellite software or operations. To handle this, OPS-SAT has to be robust and safe. This thesis describes the aspects that make OPS-SAT robust hardware wise and specifies a flexible framework, to test experiments on ground, based on GitLab. The framework is able to semi automatically handle all 115 proposed experiments so far and is able to grow with more registrations. It can create logs and regression views for the operators at ESA to see the development of the experiments over time. Initial tests for file size which create reports for the experimenters are possible. The tests are automatically scheduled. Also, the experimenters can run their own software on duplicates of the experiment processing platform at European Space Operations Center (ESOC) in a future iteration of the project.

Kurzfassung

Das Testen von Innovationen im Bereich der Missionsplanung und Durchführung gestaltet sich schwierig, da kein Betreiber eines aktiven Satelliten einen Eingriff in sein System erlauben würde. Dies wäre zu riskant, denn falls die Austauschsoftware fehlerhaft ist, verliert man einen eigentlich funktionierende Satelliten. Ein weiteres Handicap von Satellitenmissionen ist, dass die Hardware strahlungsresistent sein muss. Die Computerchip-Industrie verdoppelt ungefähr alle zwei Jahre das Rechenvermögen von Computerchips, und die Herstellung von Computerchips die strahlungsresistent sind kostet viel Zeit und Geld. Die strahlungsresistenten Chips können mit dieser Verdoppelung nicht mithalten, weshalb immer mehr kommerziell hergestellte Produkte Einsatz in Satelliten finden. Jedoch hat der Einsatz von kommerziell hergestellten Computerchips im Weltraum auch Nachteile. Nur durch spezielle Verfahren, wie etwa Redundanz, kann man die Lebensdauer eines Satelliten mit kommerziellen Computerchips an die eines Satelliten mit strahlungsresistenten Chips angleichen. Die weitere Erforschung des Einsatzes von kommerziell hergestellten Chips in Satelliten ist eine Herausforderung.

Die ESA stellt sich diesen zwei bekannten Herausforderungen mit dem neuen Satelliten namens OPS-SAT. Dessen Hauptaufgabe ist es, für neue Experimente im operationellen Bereich ein fliegendes Labor zu sein. Jede europäische Institution oder Firma kann Innovationen im operationellen Bereich von Satelliten testen. Die Experimente sind in der Lage, den gesamten Satelliten zu steuern, obwohl ihre Erschaffer geringe oder keine Erfahrung in der Softwareentwicklung und Satellitensteuerung haben müssen. Um mit eventuellen Fehlern zurecht zu kommen, muss OPS-SAT robust und sicher sein. Diese Arbeit beschreibt die hardwareseitigen Aspekte, die OPS-SAT robust und sicher machen und spezifiziert ein flexibles System um Experimente am Boden zu testen, basierend auf GitLab. Das System ist in der Lage alle 115 Experimente die bis jetzt registriert sind semi-automatisch zu testen und ist erweiterbar. Das System erstellt Logs und Regressionsdaten. Validationstests, so wie zum Beispiel das Einhalten der maximalen Dateigröße sind möglich und werden dokumentiert. Die Tests werden automatisch zeitlich geplant und ausgeführt. Außerdem können die Teilnehmer in der Zukunft ihre Experimente auf einem der Duplikate der Experimentier-Plattform des Satelliten am Boden testen.

Contents

1. Motivation **9**
- 1.1. Motivation . . . 9
- 1.2. Scope of this Work . . . 10
- 1.3. Contents of Work . . . 11

2. Technical Background **13**
- 2.1. Challenges . . . 13
- 2.2. OPS-SAT Description . . . 15

3. Related Work **19**

4. Conceptional Work **21**
- 4.1. Error Sources Throughout the Life of a Satellite . . . 21
  - 4.1.1. Hardware Before Flight . . . 22
  - 4.1.2. Hardware in Flight . . . 23
  - 4.1.3. Software Before Flight (Operators) . . . 24
  - 4.1.4. Software Before Flight (Experimenters) . . . 24
  - 4.1.5. Software in Flight (Operators) . . . 24
  - 4.1.6. Software in Flight (Experimenters) . . . 24
  - 4.1.7. Operations in Flight (Operators) . . . 25
  - 4.1.8. Operations in Flight (Experimenters) . . . 25
- 4.2. Architecture for Experiment Testing . . . 26
  - 4.2.1. Automated Testing . . . 31
  - 4.2.2. Version Control . . . 31
  - 4.2.3. Data Transfer . . . 32
  - 4.2.4. Initial Tests Defined by ESOC . . . 32
  - 4.2.5. Experimenters Tests on ESOC Hardware . . . 33
  - 4.2.6. Scheduler for Experiments on MityARM . . . 33
  - 4.2.7. Scheduler for Experiments on Flatsat . . . 33
  - 4.2.8. Experimenter Interface . . . 34
  - 4.2.9. Security . . . 34
- 4.3. GitLab . . . 35

4.4. Flowchart for Testing 36

5. Practical Implementation **39**
5.1. Configuring GitLab on ESOC Hardware 39
5.2. Interfaces 40
5.3. Testing procedure 42

6. Results **47**
6.1. Recapitulation of Assigned Task 47
6.2. Realization of the Architecture 47
6.3. Validation Run of Testing an Experiment 48

7. Conclusion **57**
7.1. Summary 57
7.2. Future Work 57
7.2.1. Email Notifications 57
7.2.2. Testing on MityARM 58
7.2.3. Automatic Data Transfer Server 59
7.2.4. Runtime Flight System 59
7.2.5. Battery Model 60

A. Installation Process **I**

B. Flowchart for Experiment Test **V**

List of Figures **VII**

List of Tables **IX**

List of Listings **XI**

List of Acronyms **XIII**

Bibliography **XV**

1. Motivation

1.1. Motivation

Space agencies and satellite operators are handicapped by the fact that the OBCs are not on the cutting edge of what technology has to offer. Making radiation hardened parts takes time and costs more money than using COTS parts. So, by the time the satellite is launched, the on-board computers can easily be 6 years or more out of date. Also, when in flight, it is almost impossible to test new procedures in satellite operations, because the risk of losing a healthy satellite when a test goes wrong is too high.

The European Space Agency is addressing those and more challenges in its new flying laboratory, a satellite named OPS-SAT. It is devoted to demonstrate the drastic improvements that will be possible, if there would be up to date computers aboard of satellites. Even though it is only 30 centimeters tall, its on-board computers are 10 times more powerful than any current ESA spacecraft[13]. The difficulty to perform live testing of mission control systems gets lowered, by having a small, cost effective satellite. This fact enables ESA to build an exact copy of the satellite. This copy is called the "Engineering Model" or "Flatsat". Experiments can be tested on it before being uploaded to the satellite. With OPS-SAT, no other mission has to risk their working satellites in orbit. OPS-SAT is build safe and robust, ready to recover from events that would usually mean the loss of a mission. With this confidence, the operators are able to try out new and innovative control software submitted by experimenters. Achieving this level of confidence and safety for a low cost project is not easy. The Team creating OPS-SAT is able to do so, by combining off-the-shelf solutions that are already common in the nanosatellite sector, terrestrial microelectronics for the on-board computer, and the experience of ESA in safely operating satellites. The result is an open, robust, cost effective 'laboratory' for in-orbit demonstration of revolutionary new control systems and software, that would be too risky for trial on a satellite in operation. Over 100 Experiments are already registered to fly on the satellite.

To maximize the on-board testing time for experimenters, as well as to minimize the risk of losing the satellite to unforeseen malfunctions of the experiments in flight, there will be a need for extensive pretesting on the ground. With those tests, it will

also be possible to compare in-flight behavior of the experiments to characteristics that were evaluated on the ground. To do this for the 100+ already registered experiments, that are constantly evolving, is already too much to manage manually. Especially under low cost conditions. This thesis is an attempt for the specification of a safety framework and the partial implementation in the OPS-SAT project.

1.2. Scope of this Work

The task proposed by ESA and Embedded Systems and Applications Group (esa) at the TU Darmstadt was to specify and partially implement a test and control framework for the OPS-SAT mission. The specific requirements cover:

1. Handling all experiments registered so far as well as experiments that will be registered in the future

   The solution shall have the ability to grow with the number of registered experiments.

2. Minimize risks for the spacecraft in flight

   Adding up on the safety features of OPS-SAT that are already in place, experiment tests performed on ground will help decrease the risk of losing the satellite.

3. Maximize experiment time in flight

   By enabling experimenters to test their own experiments on the engineering model of OPS-SAT, they can evaluate their own software and thus have an additional debugging instance.

4. Create standards for experiments

   There are several regularities for OPS-SAT. For example the file size has to be limited, because of the short communication windows that will be available in the planned orbit.

Excluded from specification in this thesis are the hardware components of the satellite, because, at the point the thesis started, the hardware of the satellite was already decided upon. The hardware will be described nevertheless for its error avoidance abilities in Chapter 4.
Boundary conditions and optimization goals are identified as follows:

1. Internal *safety and security* procedures that are necessities by ESA.

2. With OPS-SAT being a low budget mission, the *cost* shall be minimal.

This shall be realized by specifying an architecture that includes the following entities:

Entities involved in test and control architecture

1. *Experiment Developers*, that submit the code to be executed on the satellite.
2. *Satellite Operators*, who provide boundary conditions for satellite hardware and software.
3. *Offline test suits*, to be executed before the upload of an experiment to the satellite.
4. *Log and regression view*, to provide feedback about executed online and offline test results as well as results of experiments to experiment developers and satellite operators.
5. *Communication*, ensuring proper uplink of experiment software and downlink of experiment and test results
6. *Runtime flight system*, to provide online checks of boundary conditions not to be exceeded by the experiments.

1.3. Contents of Work

The technical background provided in Chapter 2 is required to understand the significance of this work, as well as to introduce the OPS-SAT project. The introduction starts with a description of OPS-SAT, its mission, and the events in the space sector that created a need for new approaches to creating satellites.

A literature research will be provided in Chapter 3. The literature research provides an overview of why COTS components are getting more and more common in satellite operations, and lists a similar mission to OPS-SAT.

Chapter 4 describes a methodical approach to identify a solution for the task of this thesis. Steps included in this approach are, abstraction of the challenge, identifying sub challenges, identifying sub solutions for challenges, creating a rating system for each sub solution to find the best ones and creating a solution for the whole project from all the sub solutions. The whole process will be explained and the outcome described.

Chapter 5 shows the practical implementation of the solution into the ESOC infrastructure. This is done by following strict security measures of ESA.

Chapter 6 provides an overview of what the solution is capable of, and shows the results of a validation test.

In Chapter 7, all those findings will be summarized and an outlook for the future of the project will be identified.

The usual Discussion to compare results to consisting solutions is intentionally left out, as there is no comparable mission or predecessor.

2. Technical Background

2.1. Challenges

OPS-SAT has two main objectives: Tackling the challenge of OBC being out of date, even at launch day and the need for a test platform for new operational methods.

The OBC being out of date roots in the harsh environment of space itself, where there are several environmental influences that can alter a satellites behavior. The main influences on a satellite in orbit being, *gravity, electromagnetic radiation, atmospheric interference, the ionosphere and matter particles.* All of them can interfere with a satellite.

The OBCs are mostly influenced by electromagnetic radiation. When a charged particle collides with an element of a computer chip it can cause a variety of failures, that can be grouped in the categories. Failures because of total ionizing dose accumulation, neutron or proton flux events and single event phenomena are a selection of them. The table Table 2.1 from [7] shows the main categories of radiation environment failures, and typical effects on electronic components. For all effects mentioned, there is some shielding feasibility.

Common measures against radiation related failures are, to avoid high radiation areas by choosing other orbits, shielding to lower the radiation dose level, use of radiation hardened components or system level error corrections. Lately, a trend in space industry is to use more COTS products, that have none of the above qualities.

It is easy to understand, that the regular approach of developing a radiation hardened chip takes a lot of time, effort, and money. This leads to cost intensive hardware, that is years behind state of the art computer chip technology.

The approach, using COTS technology in space has its upsides and downsides as well. By using several of the same COTS hardware components it is possible to get the same lifetime as using radiation hardened hardware. This can be done in cold redundancy, where the secondary component will only switch on in case of a failure, or in parallel, which enables a majority vote for computations. On the other hand, COTS hardware is not as rigorously tested and does not have the same documentation standards as specially designed space hardware. This makes it harder to make predictions for when they will fail. For example, without shielding,

Radiation environment	Typical effects
Total ionizing dose	• Threshold shifts in CMOS transistors, leading to failures of logic gates • CMOS field-oxide charge trapping, loss of isolation, excessive power-supply currents • Power transistor threshold shift, loss of on/off control • Gain degradation in bipolar-junction transistors
Neutron or proton flux events	• Displacement damage effects • Gain degradation in bipolar-junction transistors • Severe degradation of charge-coupled devices, dynamic memory performance • Damage to photodetectors
Single-event phenomena	• Single heavy ion causes ionization “track” • Single bit errors in static memories • Localized latchup in CMOS integrated circuits • Gate rupture of power transistors • Temporary upset of analog devices such as amplifiers • Burnout of diodes, transistors

Table 2.1.: Radiation related failures of electronics in space from [7]

the total ionizing dose that accumulates can lead to gain degradation in bipolar junction transistors.

In conclusion, COTS components can be a way of introducing massive computing power into satellites and space missions, but their lack of documentation and testing makes them unpredictable. Studies of COTS components in space have lead to several solutions for maximizing success rate in space. For example cold redundancy or majority votes.

The lack of testing facilities for spacecraft operations is also an issue in spaceflight. No active mission will allow experiments with their satellite, because the risk of losing it is too high.

2.2. OPS-SAT Description

OPS-SAT, with its powerful COTS OBC will be an in-orbit testing facility for ground breaking new technology. While being a three unit cubesat only 30 centimeters tall, OPS-SAT combines robustness, safety and more computational power than any other mission, into a flying laboratory for innovations in the mission operations sector.

Its robustness allows the satellite to provide open access for registered experimenters, that have ideas for new methods in space craft operations. Experimenters can upload their experiments and test how these perform in the conditions of outer space.

OPS-SATs mission objective is to be an orbital test platform for *on-board software applications, advanced communication protocols, compression techniques, demonstration of advanced software-defined radio concepts, optical communication from ground to space, experiments using cameras, attitude control, scheduling and autonomy as well as experiments with ground-based applications.* OPS-SAT is packed with state of the art semiconductors to fulfill its mission objectives. This includes processors, field-programmable gate arrays (FPGAs), a camera and an attitude determination and control system (ADCS) for pointing to exact locations and recovery of the satellite in safe mode. Experimenters will be able to access many sensors and actuators, such as reaction wheels and magnetorquers, either via an application programming interface (API), or directly via telemetry. The OBC will relay the signal in telemetry mode during the experiments execution.

The OPS-SAT architecture has two major parts, the OPS-SAT bus and the payload, as seen in image Figure 2.1. A satellite bus is the infrastructure of the satellite, that provides a base for the payload. The bus of OPS-SAT consists of four entities, namely *electrical power system (EPS),on-board data handling (OBDH), coarse ADCS and telemetry and telecommand (TMTC).* The EPS is responsible for power input, power output, battery and solar panels. The solar panels are used to charge the battery, which can provide power even when the satellite is in earths shadow. The EPS also monitors the power consumption of the components of OPS-SAT. If the power consumption of one of the components is too high, the EPS will open the switch. This is one criteria for shutting down experiments.

The OBDH consists of the two main OBCs, which are in cold redundancy. They are in charge of managing the OPS-SAT modes, reading the coarse attitude sensors, loading, monitoring and managing the executables into the payload computer, executing telecommands, storing and sending the telemetry messages to ground and obtaining the GPS time and distribute it to all computers. Also, the OBC collects sensor data for all bus and payload devices and with that monitors the satellites health. It also performs adequate recovery actions according to its fault detection, isolation and recovery (FDIR) functionality. This includes receiving high priority commands sent via ultra-high frequency (UHF).

Coarse ADCS are used to get information about the satellites attitude in orbit. They are not sufficient enough for pointing, but in case of an emergency, or when a coarse attitude reading is needed they will be used.

TMTC includes everything that is needed to communicate with the satellite.

The core payload consists of the satellite experimental processing platform (SEPP), S-band TMTC and the ”consultative committee for space data systems (CCSDS)” engine. S-band TMTC is used for faster uploading data, such as experiments and downloading telemetry produced by the experiments. The CCSDS engine manages the data transfer as well as telecommunications and telecommands according to the CCSDS requirements.

The SEPP is the main experimentation platform. It is based on an Altera Cyclone V SX System-on-Chip(SOC) digital core logic device, which is a COTS component. It provide powerful processing capability with its 800MHz central processing unit (CPU) and 1GB DDR3 RAM. Four of those boards will be on board in cold redundancy. The board consists of a hard processing platform (HPS) and an FPGA portion. The HPS is a fully functional computer that contains a dual core CPU with hardware blocks and device interfaces. The HPS is connected to FPGA and able to communicate with it. The FPGA consists of logic array blocks (LABs), memory logic array blocks (MLABs) and arithmetic logic moduless (ALMs). By connecting these blocks using very high speed integrated circuit hardware description language (VHDL), it is possible to create custom logic structures.

The SEPP has Linux as default operating system. For OPS-SAT, experiments can be deployed in of four ways: As a Java virtual machine Java virtual machine (JVM), that uses the Nanosat MO-Framework to access the sensors and actuators, as well as the camera of the satellite, onto the Linux operating system (OS), as binary on the kernel, of the system or as FPGA firmware as mentioned in [9].

The first option is, to create a JAVA application that uses the Nanosat MO Framework. The MO Framework, according to [11], is an innovative on-board software framework for nanosatellites. It is able to provide a way to easily interact with a nanosatellite. This is possible by introducing the concept of an "App" for a mobile phone to a nanosatellite. The apps can be easily developed tested and deployed. Even execution on multiple spacecraft is possible. It is based on the CCSDS communication layer for satellites. The API enables experiments to access the sensors and also to interact with the TMTC part of the satellite.

The second option is to develop software to be installed directly onto the Linux OS. This software does not necessarily have to be written in JAVA. By developing a software to run on the Linux OS, it is possible to replace the protocol stack. This is interesting for developers that want to test innovations in data transfer from satellite to earth.

Thirdly, there is a way to install the experiment raw on the kernel of the SEPP. This is useful for experiments that want to test a completely new OS.

Lastly, a custom FPGA firmware can be deployed to the satellite.

For the installation of an experiment, it will be uploaded as a binary file together with an installation script. The script then installs the experiment on the SEPP. As a safety measure, the experiment have to send an "alive" signal periodically to the main OBC. This way it is possible to recognize when en experiment does not work properly.

When a part of the satellite is not functioning correctly, it is beneficial to trigger a "safemode" command, to keep the satellite safe. In safemode, only the main components, namely the satellite bus stays operational. This way, the satellite saves power until it is restored by the operators to complete functionality. Also, the satellite starts in safemode, when it is switched on. The possibilities to trigger a safemode are among other things, if the EPS registers an abnormal power usage, when the ADCS exceeds its tumbling limits, or with a reset command from earth.

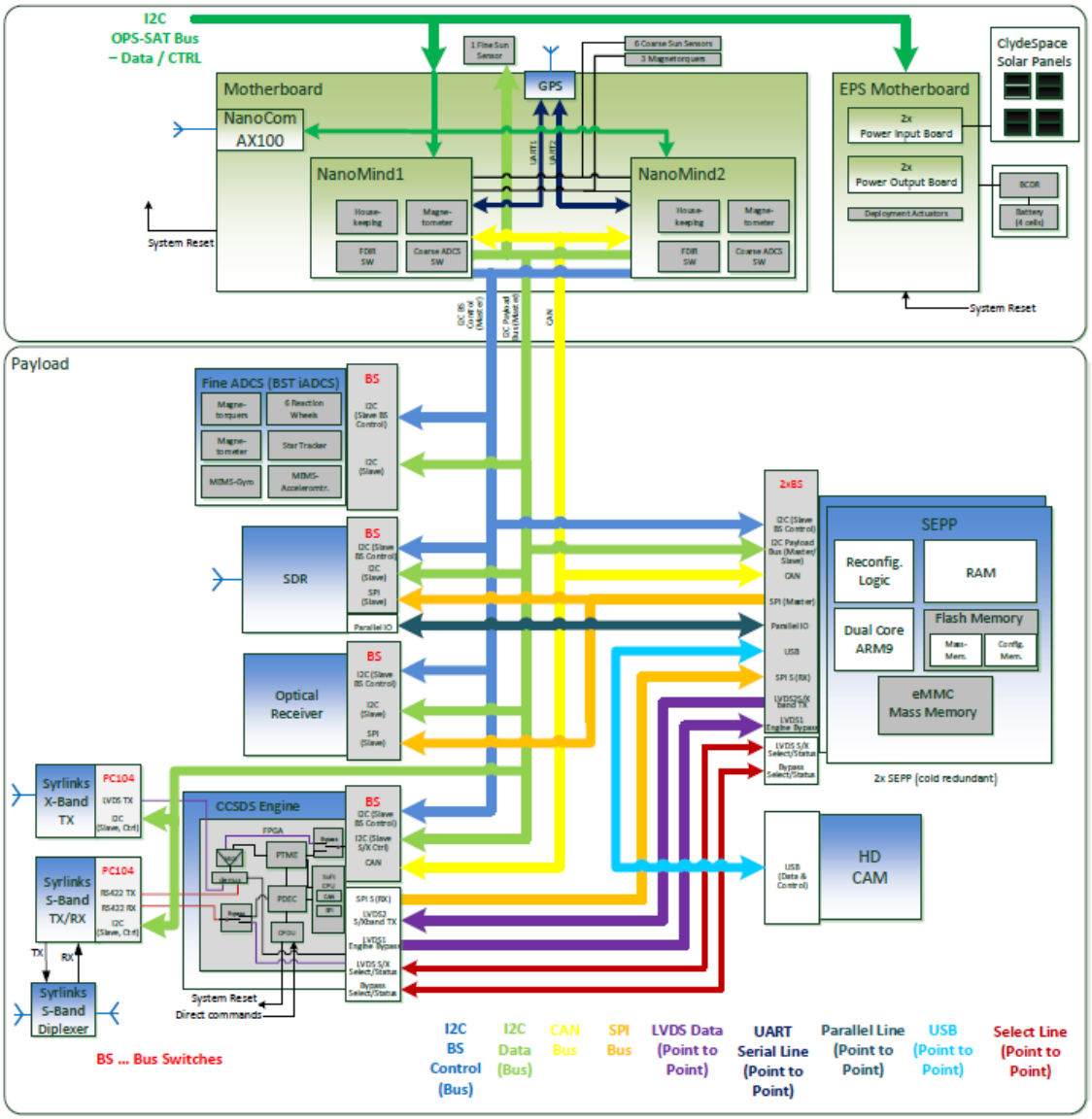


Figure 2.1.: OPS-SAT Architecture [12]

3. Related Work

This chapter will briefly outline, why COTS components are getting more and more used in satellite production, and shows a satellite similar to OPS-SAT. COTS components have been used in spacecrafts from the first scientific missions to the new innovative small satellite programs of the 21st century [3]. The cost reduction paired with the ever growing computing power makes it intriguing to use COTS products. Using them is only possible by applying rigorous tests to the parts, to gather data about their behavior in the natural space environment, described in Chapter 2. Those screening procedures are pricey, so studies work on standardizing procedures to safe money. National Aeronautics and Space Administration (NASA)'s [14] paper acts as a guide to use appropriate parts in the fabrication of space hardware, to meet mission reliability objectives within budget constraints. OPS-SAT further tests the abilities of COTS components in space.

OPS-SAT, as a flying laboratory to test new approaches in COTS hardware, as well as operational methods, is the first of its kind. There was a mission planned to test COTS hardware, NASA Space Technology 8 (ST8) mission, set to launch in 2009, but has been cancelled [1]. Especially the exchange of experiments in flight, that OPS-SAT provides has never been done before.

4. Conceptional Work

This chapter provides a simple overview of error sources during the life of a satellite and error avoidance strategies. Simple, because error avoidance in satellite missions is not the main objective of this thesis, but nevertheless a part of it. After the overview, the focus will lie on the identification of partial challenges for testing experiments provided by experimenters. Those partial challenges are analyzed and rated, before combining them into one solution for testing experiments.

4.1. Error Sources Throughout the Life of a Satellite

One of the biggest challenges for the OPS-SAT development is to build a satellite that can be controlled by amateurs, with little to no experience in spacecraft operations and be robust enough to not lose the mission when an error occurs. Not losing the satellite shall be the main objective of the hereby proposed safety and control framework.

Errors in a satellite mission can occur either before flight or in-flight and can be caused by hardware, software or the operation of the satellite as such. Satellite operations cover all activities that are necessary to control and maintain the satellite in working condition. In OPS-SATs case, there has to be another differentiation, because experimenters are allowed to control the satellite. Experimenters do not necessarily have experience in spacecraft operations, nor in software development. Experiments are software that are controlled by the experimenters, therefore errors produced by experimenters can be of software or operational nature. This means, additionally to operators as error sources, we have experimenters in the categories software before flight, software in-flight and operations. Section 4.1.1 outlines the whole framework with its error sources, but will focus on the error avoidance for software before and in-flight for experimenters.

As mentioned before, the errors that can occur in hardware before flight and hardware in flight are just mentioned here, because the decisions for hardware were already made when this thesis started. Nevertheless, the test and control framework will include a description of measures to take to prevent some hardware errors both

before and in-flight in the Section 4.1.1. A complete detailed error analysis for the satellite would go beyond the scope of this thesis.

4.1.1. Hardware Before Flight

First, a selection of hardware errors before flight and what can be done to avoid them, according to [5]. The spacecrafts hardware consists of structural components, sensors, actuators and the payload. The structure of the spacecraft is most likely a lightweight material, that still can resist the immense forces that influence the spacecraft in its life phases. First, the satellite has to withstand the vibrations and acoustical stress that happens during the launch of a rocket. When released into space, the satellite gets battered with solar and cosmic rays, eroding any material that has not been qualified for space environment. Micrometeorites, blasting through space at mind blowing speeds can crash into the structural material, destroying any part that has not been rated for impacts. Also, the temperature gradients, caused by the combinaton of solar radiation and thermal vacuum, can lead to stress in a material, resulting in a deformation or fractures. To avoid structural errors, extensive modeling and testing of all the structural forces is needed. Tests include acoustical testing, vibration tests, radiation tests and so on. A recent famous representative of a structural error causing a spacecraft to fail was the explosion of Falcon 9 Rocket 2015. One of the pressurized Helium tanks ruptured, destroying the rocket and the payload [15].

Sensors on a spacecraft are needed to determine its spacial information. Not knowing a spacecrafts altitude, attitude or orientation towards a reference system, can result in the loss of the satellite, for example caused by communication loss when directional antennas are pointed in the wrong direction. There are several main sensors to determine a spacecrafts attitude. For example Gyroscopes, startrackers, magnetometers, accelerometers and tracking with radio waves using beacons, gps and doppler radar. Possible error sources for the sensors in a satellite can be human error such as in the crosswiring of gyroscopes in the Genesis mission 2001. The satellite crashed at landing, because the accelerometers had been installed backwards. Also, the Galileo Jupiter entry probe had a fatality when the parachute deployed 1 minute too late. Also due to crosswired accelerometers. The fault had not been noticed, because the probe was tested in a centrifuge, which had a crosswired test harness [5]. Other error sources include faulty calibration or interface problems. To avoid human errors, management systems should be in place as well as the right procedures to handle sensors.

Actuators are needed, to keep the spacecraft on the correct track to its destination, or to control a spacecrafts attitude. Common actuators in spaceflight are thrusters and reaction wheels. Thrusters work with the principle of ejecting accel-

erated matter, and the resulting acceleration of the spacecraft. Most thrusters work with pressurized substances, that get ejected through a nozzle. To contain and regulate the pressure, valves are needed. The valves have many attack vectors for errors, such as the mostly corrosive materials used for propulsion, which include Oxygen, Kerosene or Hydrazine. The valves can get stuck, or break, caused by corrosion. Solutions for stuck valves can be the parallel arrangement of valves, if the weight restrictions allow it. For the corrosion of the valves, a serial arrangement could be made. An example for a mission that had issues with its valves is the tracking and data relay satellite (TDRS) 9. The US satellite was launched on an Atlas rocked and and should insert itself into a geostationary orbit. The first burns went well, but then the sensors indicated that no fuel was flowing. One of the pressurization valves was stuck and the fuel can not exit its tank without pressurization. Fortunately, the pressurization lines were redundant, what saved the mission [5].

Reaction wheels are used to control a spacecrafts attitude. But they too are prone to errors. For example, two of the reaction wheels of NASA's Kepler "Planet Hunter" satellite failed after only two or three years in orbit, respectively [8]. Luckily, Kepler could be resurrected using the pressure of solar winds to stabilize the spacecraft without two of its reaction wheels.

The payload of a satellite is usually not a major threat to mission health. But a payload that does not meet the safety requirements could become an error source. The payload can be a sensor, a structure, or an actuator, so all the safety measures stated above have to be applied to the payload of a satellite.

4.1.2. Hardware in Flight

A second source of errors is hardware in flight. As discussed in Section 2.1, the radiation from the sun and the cosmic background radiation can alter a spacecrafts structure, which has to be chosen according to the expected environment. But not only structure can be influenced, silicon based computer chips are also at risk of failing. In Table Table 2.1 on page Page 14 some typical effects of different radiation environments are shown. Traditionally such errors are prevented by using radiation hardened computer chips. Those are developed especially to withstand the radiation in space. When using COTS components, there are several approaches to reach the same range of lifetime as with radiation hardened components. As mentioned in [3] error reduction methods are, redundancies, component inspection and testing in a thermal vacuum facility. The testing of components in thermal vacuum facilities is used to provide data about their lifespan in conditions for which they were not made.

4.1.3. Software Before Flight (Operators)

As for every other field of application, satellite software development is also error prone. For example, exception handling and knowledge of system boundaries have been sources of errors in the past. The loss of the first Ariane 5 rocket was due to a specification and design error in the inertial reference system[2]. Or the famous omission of a hyphen that allowed incorrect transmission of guidance signals to the Mariner 1 Probe, causing in a fatal accident [10]. There are many possibilities for error correction, such as unit tests, test driven development, modular programming or tools for code analysis [4].

4.1.4. Software Before Flight (Experimenters)

Besides the more experienced operators, experiment developers may cause software errors during development (i.e., before flight) in the OPS-SAT project. Tests on ground can be carried out, to rule out any errors before the flight. Potential tests could be initial tests performed by the satellite operators, or logging performance data of experiments, such as CPU or random access memory (RAM) usage of the experiment on the ground. Also, enabling experimenters to test their own software on the engineering model of the satellite can help detect software bugs. This section will be focused on in Chapter 7

4.1.5. Software in Flight (Operators)

When the satellite is in its orbit, software errors can also occur. Either, bugs that have not been eradicated before launch can be detected, or, when a software change is needed, new bugs can be introduced. Possible reasons why a software change is needed are on-board failures or the before mentioned detection of bugs. Factors that can cause the introduction of errors in flight are time pressure or the team not knowing their system perfectly. To avoid the errors mentioned here, a bug fix that is not system critical should be planned ahead with enough time to rule out every possible error. To rule out errors that could be caused by a team not knowing their system perfectly, internal changes in teams should be limited, so that no knowledge gets lost with a change. Also, extensive documentation of produced data is possible to minimize errors.

4.1.6. Software in Flight (Experimenters)

Experiments will be changed on a regular basis, that of course introduces a lot of possible error sources. Inexperienced experimenters can introduce errors that can

possibly influence the spacecraft. For example, when an experiment uses the CPU of the experimenters platform excessively for a long time, the battery could be drained. There is already a hardware way to stop an experiment and send the satellite to safe-mode. The EPS recognizes if any unit of the satellite, thus also the experiment unit, uses too much power and shuts down the corresponding section. This allows the satellite to recharge its batteries. Also, the experiments have to send an alive signal periodically [9]. When the signal is not recognized by the main OBC, the experiment also is shut down. A software solution that is able to warn the experiment, that it exceeds its limits for resource usage would be a more elegant solution. This way, the experiment can go on and check what is not functioning correctly, or save the data produced so far. By logging the performance data of the experiment before hand like proposed in Chapter 7, it would be possible to create unique and exact limits for every single experiment. Thus having a way of comparing the resource usage on orbit with a terrestrial testrun.

4.1.7. Operations in Flight (Operators)

Yet another error source can be the decisions that satellite operators make throughout the mission. For example when an attitude change with wrong values is uploaded and the satellite is far away, this could lead to the high gain antenna pointing away from its terrestrial counterpart, thus losing communication and triggering a safe mode. To avoid such mistakes, teams could double-check every decision, or have automated tests in place before uploading operational commands.

4.1.8. Operations in Flight (Experimenters)

In-flight, there is another threat to the satellites health, namely the operational decisions that the experiments make. One operational mistake that could happen is, for example, the CPU of the experimenters platform runing on high energy consumption in the eclipse of the orbit. Or, when an experiment is spinning the satellite, so that its solar cells point away from the sun, preventing it from charging. Those two actions are possible and will not be checked for by a software test. This is why the hardware of the satellite has to account for such mistakes. Measures against operational errors by experimenters are for example, that the every face of the satellite is coverd with solar cells, enabling it to charge even while spinning. Also, the EPS checks if the battery charge is getting too low, and when it happens shuts down the experiment before depletion.

4.2. Architecture for Experiment Testing

As mentioned in Chapter 3, what makes OPS-SAT different from other missions is the fact, that experimenters will also be able to control the satellite. This is the key mission element and has not been realized before by any other satellite mission. But there is also a lot of risk introduced when amateurs can control a spacecraft. Some examples of what can cause errors when experimenters control the satellite have been mentioned in Section 4.1. In the following section this thesis will describe an architecture to rule out many errors that could be introduced by experimenters handling the satellite.

The hardware to test experiments on the ground consists of five of the MityARM processors running in the satellite experimentation segment, and an engineering model of the satellite. The MityARMs will be used for experimenters to test their software for bugs before it gets uploaded to the satellite. Also, because of the small and relatively cheap satellite, there will be an exact copy of the satellite in ESOC. This engineering model, also called "flatsat" will also be used to test experiments before uploading them to the satellite in orbit. Again, the hardware was decided upon, before this thesis was produced.

The following subtasks are covered by the proposed framework. It should have automated testing, as the experiments are already too many to handle them manually. It should have version control, enabling the experimenters as well as the operators at ESOC to log the progress of the experiment. The data transfer has to be arranged, first, from experimenter computers to ESOC data storage. Then, from ESOC data storage, to test on the MityARMs and, when those tests passed from the data storage to testing on the flatsat. The architecture should be able to run initial tests proposed by ESOC on the experimenters hardware. Also, the experimenters should be able to test their own experiments on ESOC hardware, to minimize errors. A scheduler should be implemented, to organize experiments to be tested on the MityARMs and the flatsat, as there are many experiments and not much testing hardware. There should be an electronic interface for experimenters to interact with and, of course, ESA security measures are mandatory.

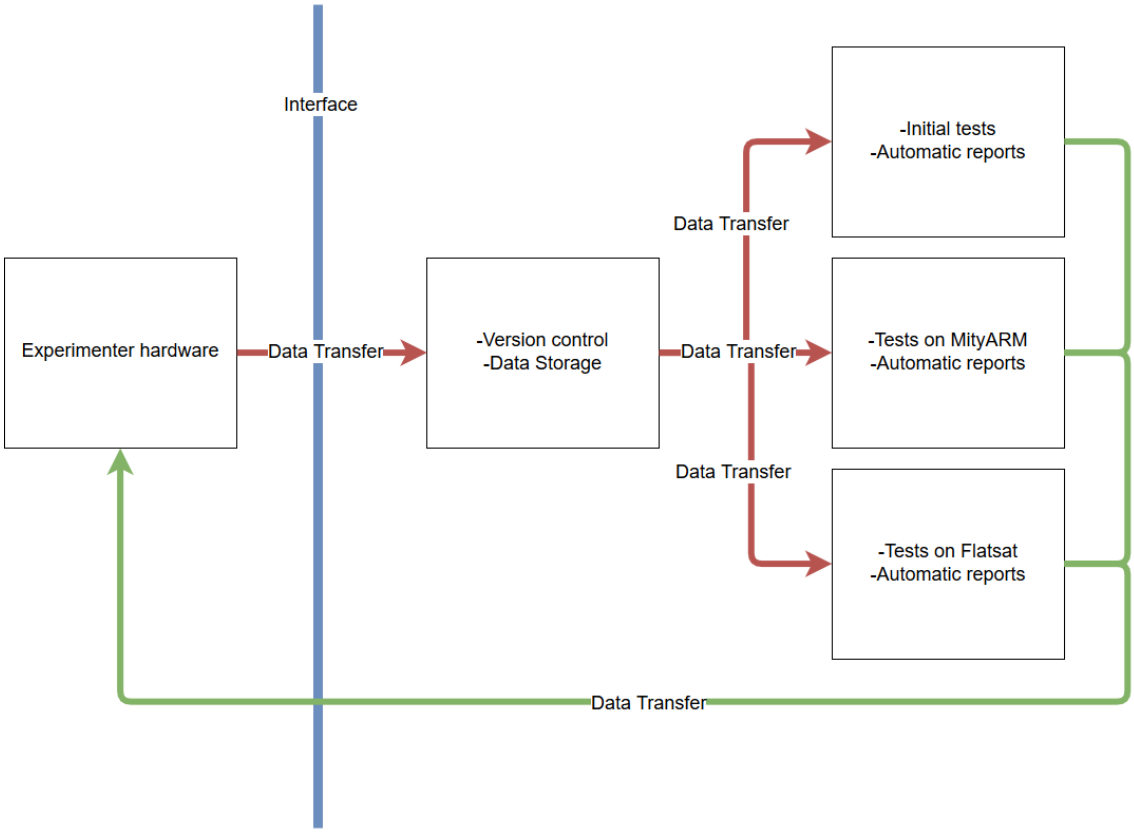
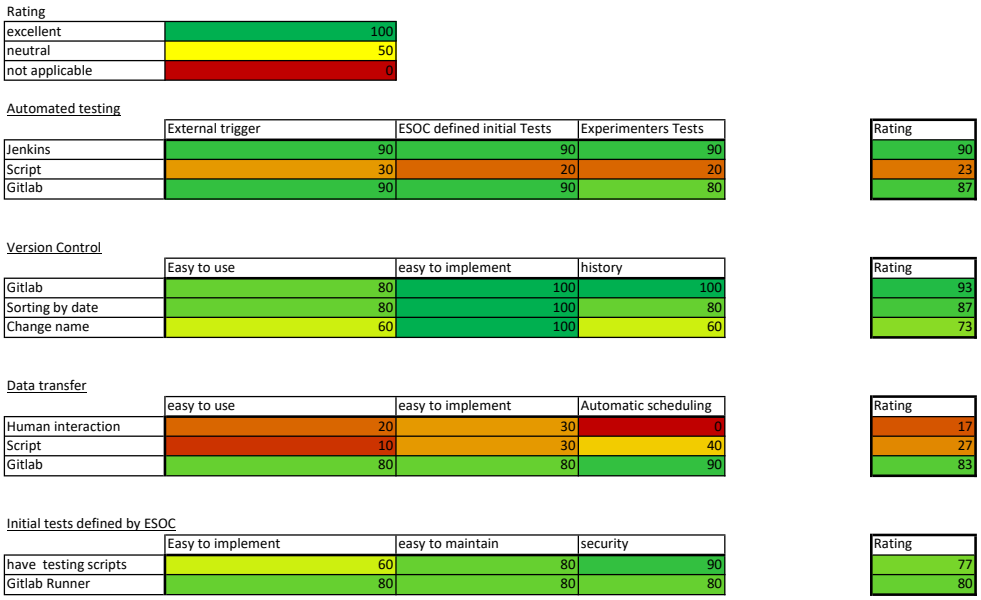


Figure 4.1.: Experiment architecture overview

Figure 4.1 shows the basic entities of the envisioned architecture. Rectangular boxes represent active entities, such as the experimenters, an instance for version control and data storage, or the entity for the initial tests. Red lines indicate data transfer of experiment data, such as binaries and scripts, green lines indicate results that have to reach the experimenters. The blue line indicates the interface between the experimenters and the testing architecture.

For those sub challenges, sub solutions were identified. Those sub solutions then were rated and compared. The sub solutions found and the rating process are described in the following section. A summary of the results is provided in Figure 4.2 and Figure 4.3.



Rating

excellent	100
neutral	50
not applicable	0

Automated testing

	External trigger	ESOC defined initial Tests	Experimenters Tests	Rating
Jenkins	90	90	90	90
Script	30	20	20	23
Gitlab	90	90	80	87

Version Control

	Easy to use	easy to implement	history	Rating
Gitlab	80	100	100	93
Sorting by date	80	100	80	87
Change name	60	100	60	73

Data transfer

	easy to use	easy to implement	Automatic scheduling	Rating
Human interaction	20	30	0	17
Script	10	30	40	27
Gitlab	80	80	90	83

Initial tests defined by ESOC

	Easy to implement	easy to maintain	security	Rating
have testing scripts	60	80	90	77
Gitlab Runner	80	80	80	80

Figure 4.2.: Summary of the Evaluation Process 1

Experimenters tests on ESOC hardware

	Easy to implement	easy to maintain	security	Rating
Share MA physically	20	10	40	23
Gitlab Runner	95	95	70	87

Schedule exp. on MityARM

	Easy to implement	Automatic	Rating
Own scheduler	10	50	30
Gitlab runner scheduler	90	90	90

Schedule exp. On FlatSat

	Easy to implement	Includes schedule of Team	flexible to be auto	Rating
COTS scheduler	90	70	10	57
Manual	80	80	10	57
Gitlab runner scheduler	60	40	70	57

Experimenter Interface

	Easy to implement	easy to use	security controllable	Rating
Website	10	50	50	37
SSH	70	90	90	83
Gitlab	90	90	80	87

Security

	Compatible with ESOC infrastructure	Rating
Own server	100	100
Cloud	0	0

Figure 4.3.: Summary of the Evaluation Process 2

The following section analyses the subtasks and identifies solutions for each. For the rating, a simple non weighted percent scale is used, 0 as the lowest rating value and 100 as the best rating possible.

4.2.1. Automated Testing

For automated testing of the experiment software, Jenkins, GitLab or writing own scripts are identified as possible sub solutions. Those sub solutions were rated if they can use exernal triggers, to trigger test processes automatically, if they can run initial tests defined by ESOC, and can run experimenters own tests. 100 percent "external trigger" correspond to the functionality already in place and correctly setup, zero percent means coding from nothing. 100 percent "ESOC defined initial Tests" means the solution is in its basic form able to run the tests defined by ESOC, zero percent means no tests can be implemented. 100 percent "Experimenter Tests" corresponds to the solution already able to have the experimenters tests implemented, zero percent means no implementation is possible. Jenkins as a software for continuous integration is able to handle all three tasks with the best rating. Writing own scripts takes longer time and the integration of the scripts in the surrounding architecture is hard. GitLab features all the integration tools that Jenkins has, and can be easily connected with its surrounding architecture. Thus GitLab was chosen as a solution for the automated testing.

4.2.2. Version Control

Version control can be done for example with GitLab, sorting by date or changing the name of the files. The sorting is the simplest solution for having different versions of a file. Those alternatives were rated for simplicity of usage, simplicity of implementation and if they create a history. 100 percent easy to use means a solution has the functionality implemented and setup, zero percent means the solution has to be handled by experts. 100 percent easy to implement means it is easy to setup and to be changed, zero percent means it is not flexible. 100 percent history corresponds to every change can be logged automatically, zero percent means a change has to be done by hand and every time. GitLab has received the best rating. Even though it is not the easiest option to implement, it is easy to use and it creates a history as well as it logs changes. Again the easy interaction between the other parts of the architecture was an advantage.

4.2.3. Data Transfer

Data transfer is needed, to transfer the experiment from the experimenters hardware to ESOC for the version control, or to transfer the data from the version control to be tested on the MityARM and the flatsat, respectively. The data transfer should be easy to use, easy to implement and automatically scheduled. 100 percent easy to use corresponds to the experiment being fully automatically transferred, zero percent means it has to be transferred by hand and by an expert. 100 percent easy to implement would mean a solution is already embedded and set up, zero percent means the solutions has to be programmed thoroughly. 100 percent automatic scheduling means the programm schedules every data transfer automatically, zero percent correspond to no scheduler can be implemented. Possible options for data transfer are Human interaction (copying files) from a to b, scripts that are written to automatically handle data, or GitLab. Transferring the data manually is how the OPS-SAT team managed data transfer so far. But the 100+ experiments registered are too much to handle manually. For the scheduling to be automatic, a stack has to be in place. For example when an experimenter wants to test an experiment in the night, there would be nobody to transfer the the experiment to the test stage. Implementing a script with a ”first in first out” stack would also solve the challenge, but is not as easy to implement in the overall architecture, and takes a lot of time to code. With GitLab it is possible to register runners that automatically listen for jobs and transfer the data automatically, when the runners are registered correctly. GitLab also includes git as a way to transfer the experiments from the experimenters hardware to the included version control tool. Thus GitLab again provides the simplest solution by having all the features implemented.

4.2.4. Initial Tests Defined by ESOC

The initial tests defined by ESOC have to be easy to implement, easy to maintain and secure in a way, that the experimenter software can not harm the surrounding system. 100 percent easy to implement would be, that the solution is already available without doing anything, zero percent means everything has to be developed thoroughly. 100 percent easy to maintain corresponds to, for example a graphical user interface (GUI) with changeable values and zero percent means everything has to be done again thoroughly for any changes. 100 percent security means there is no way at all that any experiment interacts with any other part of the system, zero percent would be direct communication available for all experiments. The test scripts are not as easy to implement as GitLab, but they both are equally easy to maintain, with values changeable in the code. The security aspect is better with the scripts, as they can be associated to only on user and can run on a separate server, isolated from the rest of the framework. GitLab can also have a separate server for

running the tests. Overall, GitLab is is easier to implement, so it is chosen.

4.2.5. Experimenters Tests on ESOC Hardware

The experimenters tests on ESOC hardware can be done with sharing the MityARMs physically by sending them with a delivery company, or by having them implemented as GitLab runners at ESOC. Criteria for the initial tests are ease of implementation, ease of maintenance and security. 100 percent easy to implement means the solution is already available, zero corresponds to the solution being not available. 100 percent easy to maintain means the hardware is at the same place, and only selected people have access to it. 100 percent security means no experiment ever is able to access data of another experiment, and the experiment has no connection what so ever to the other parts of the framework. Until now, the few experimenters that are located around Europe are sharing the MityARMs physically. This solution is neither easy to maintain, easy to implement, nor safe. With the MityARMS configured as GitLab runners, every experimenter can have safe access to the MityARMS.

4.2.6. Scheduler for Experiments on MityARM

Scheduling experiments for testing on the MityARM should be easy to implement and automatic. 100 percent easy to implement means that the functionality is already available and set up, zero percent means it has to be programmed thoroughly. 100 percent automatic means the solution actively schedules jobs and executes them as fast as possible, zero percent means the scheduling has to be done by hand. Creating an own scheduler is not easy, and can only with a great amount of integration be fully automatic. The GitLab runner has this feature ready to be setup, an it automatically listens if jobs get created.

4.2.7. Scheduler for Experiments on Flatsat

Scheduling on the flatsat is connected to greater effort. The experiment has to be marked ready for testing on the flatsat by an administrator. The upload to the flatsat has to be done via a file upload server, that is connected to the S-band receiver of the flatsat and the experiment must be installed via the communication channel of the flatsat wich is connected to a communication server. Implementing this is not easy and not yet ready by the end of this thesis. Scheduling experiments for testing on the flatsat must be done by a team member, that looks after the experiment during its execution time. To fulfill this sub challenge perfectly, a scheduler must be easy to implement and must include the schedule of the team that looks after the experiment. In the future, it could be possible to at least automatically upload

the experiment to the flatsat at a specified date in the schedule. To implement this feature, the scheduler would need the ability to access the experiments and the flatsat at the same time. 100 percent easy to implement means there is a ready to use option for the challenge, zero percent means the solution has to be created thoroughly. 100 percent inclusion of the team members schedule means the schedule is already implemented, zero percent means there is no way to implement a schedule for the team. 100 percent flexible to be automatic means the scheduler has the functionality to be upgraded to automatic file transfer already, zero percent means there is no way of automatic file transfer. A COTS scheduler is easy to implement, it can contain the schedule of the team but it is not easily modified for the automatic file transfer. Doing the scheduling manually is easy to implement, it can include the schedule of the team but it is not flexible to be automate. A GitLab runner scheduler is not very easy to implement, it can not easily include the schedule of the team, but it is able to be modified to be automatic in the future. The rating of this sub solution does not influence the decision of the overall solution, as the feature is not yet ready, but it is provided as a decision making aid for the future.

4.2.8. Experimenter Interface

The experimenter interfaces have to easy to implement, easy to use both by experimenters and by administrators and security contrallable. Security controllable contains for example access rights for users. 100 percent easy to implement means the solution is already available, zero percent means the interface has to be created thoroughly. 100 percent easy to use means the experimenters can easily access the website and the administrators can easily add the experimenters to access the website. 100 percent security controllable corresponds to all features of the website can be individually customized, zero percent means every user has the same rights. Creating an own website is not easy, and takes a lot of time. Its ease of use and security control features depend on the programmer. Ssh is not as easy to use and does not include an easy GUI. It is secure in the way that every experimenter has its own personal and public key. Gitlab has a website as well as ssh already implemented, which make it easy to setup and to use. Every user can have semi customizable security access.

4.2.9. Security

For security reasons, the architecture has to be positioned within ESOC infrastructure.

4.3. GitLab

GitLab is identified as a COTS solution, that includes many of the required features by itself. GitLab is a web-based software management tool based on git. It is an evolving project with a free community edition, an enterprise starter edition and an enterprise premium edition. The OPS-SAT project uses the community edition, which inherits all the features needed. For example the community edition has powerful continuous integration tools, that enable the handling of experiments automatically. The software itself runs on a server that hosts a website with a login screen for experimenters, it has versioning tools in place, pipelines that can be triggered externally and build artifacts for the reports to the experimenters. Also, a computer can easily be registerd as a runner and so can be the MityARMs.

On this basis, a flowchart for testing the experiments is created. Section 4.4 describes this flowchart.

4.4. Flowchart for Testing

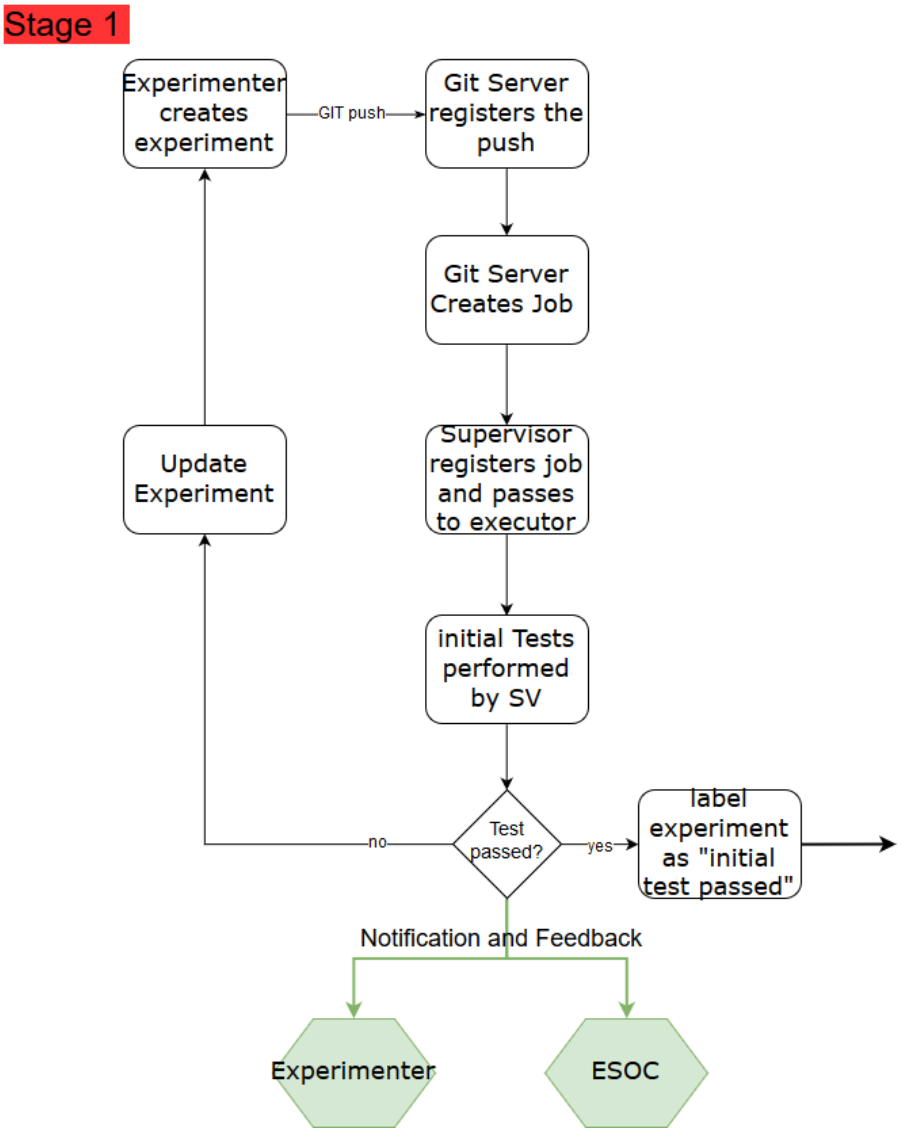


Figure 4.4.: First Stage of Experiment Testing Process

Figure 4.4 shows the first stage of theorectical procedure to test an experiment. An overview of the testing process can be found in Figure B.1 in Chapter B. First, the experimenter create an experiment. This data in form of a script and a binary is then pushed to the GitLab server that then creates a job. The job is registered by the supervisor, that passes it to the executor for the initial tests. If the test fails, the experiment has to be updated by the experimenter, if the experiment passes, it can move on to the stage where the experimenter can test their own experiments on one of the MityARMs. Either way a report is created and the experimenters as well as the operators are notified and get the results of the tests.

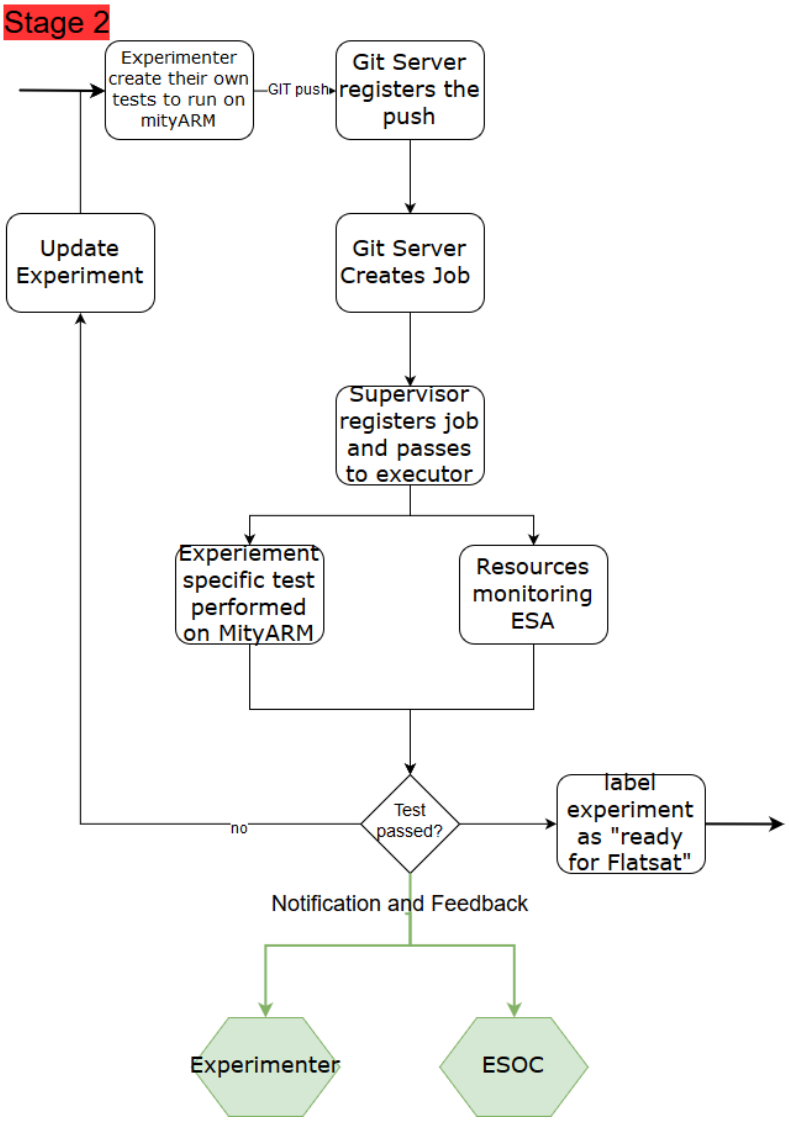


Figure 4.5.: Second Stage of Experiment Testing Process

Figure 4.4 shows the second stage of the theoretical testing process. Here, the experimenters will create their own tests to verify the correct execution of their software on a copy of the satellites experimenter platform, the MityARM. When the experimenters already created their own tests, this of course is skipped. When the data is then pushed, the GitLab server registers it and creates a job. The supervisor picks up the job and sends it to its corresponding executor. There, the experimenters can run their tests, and the OPS-SAT team can create logs of resources, to compare them to the experiment, when it is executed on the satellite in orbit. Further information in Chapter 7. The results of these test are also saved and the experimenters as well as the OPS-SAT team notified. When the experimenters are satisfied with the performance of their experiment, it can then move on to be scheduled on the Flatsat for testing in Stage 3

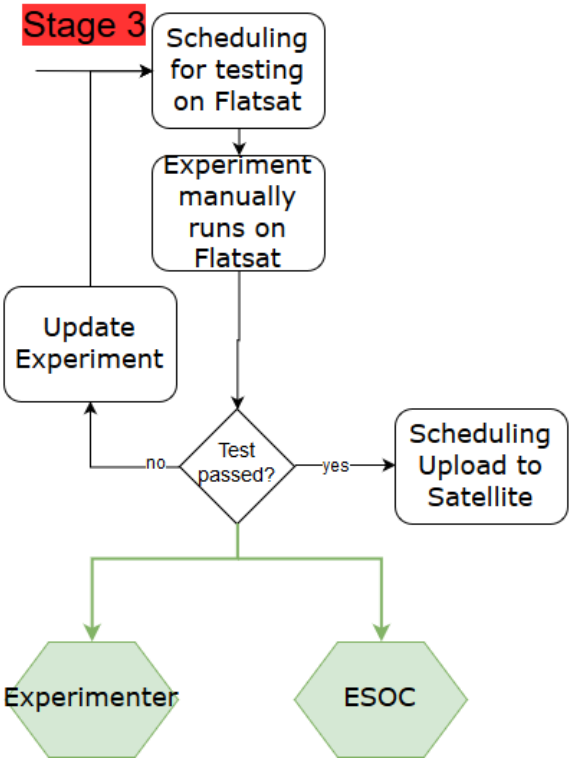


Figure 4.6.: Third Stage of Experiment Testing Process

Figure 4.6 shows the third stage of the theoretical testing process. There, the experiment gets scheduled for testing on the flatsat, and afterwards it is uploaded and executed manually. If the test passed, the experiment gets scheduled for uploading to the satellite, if not, the experimenters have to update their experiments and reschedule the test. Again feedback is created and the experimenter as well as the team get a notification.

5. Practical Implementation

This chapter describes the practical work installing and configuring GitLab on the virtual machines in ESOC. The detailed installation process of the GitLab server and the GitLab runner server is described in Chapter A.

5.1. Configuring GitLab on ESOC Hardware

The framework for testing experiments consists of five entities: Experimenters infrastructure, a firewall, the GitLab server, the GitLab runner server (supervisor) and the GitLab runners. Experimenters infrastructure refers to the experimenters personal computers (PCs), on which they create and store their experiments. The hardware structure with interfaces is visualized in Figure 5.1. This image is more detailed and adjusted to GitLab as a solution, in opposition to Figure 4.1, which shows a generel solution with only sub challenges.

On the left, the experimenters are shown. The firewall is required by ESOC to restrict the access to ESOC infrastructure from outside the ESOC network. The exact mechanism and network can not be shown due to safety reasons by ESA. Inside the ESOC network, there is the GitLab server, the GitLab runner server and the runners. The MityARM is in the network, and registered as a runner, but communication with it is not yet possible as described in Section 5.3. The file upload server and the flatsat are not yet implemented at this point. The GitLab server stores all the experiments and GitLab is installed on it. It is, among other things, responsible for creating the jobs that the supervisor can pick up. Experimenters can also see the history that git creates with their account on the website provided by the GitLab server.

The GitLab runner server acts as the supervisor for jobs created by the GitLab server. It waits until a job is created and allocates it to its corresponding runner, which executes the job. The jobs are created when an experimenter pushes an experiment to the server. The ".gitlab-ci.yml" file, that is located in the initial tests project, defines stages and tags. The stages ensure that the jobs will be executed after another, GitLab otherwise runs the tests in parallel. Parallel tests would be better performance wise, but if the initial test fails, the experimenter is restricted

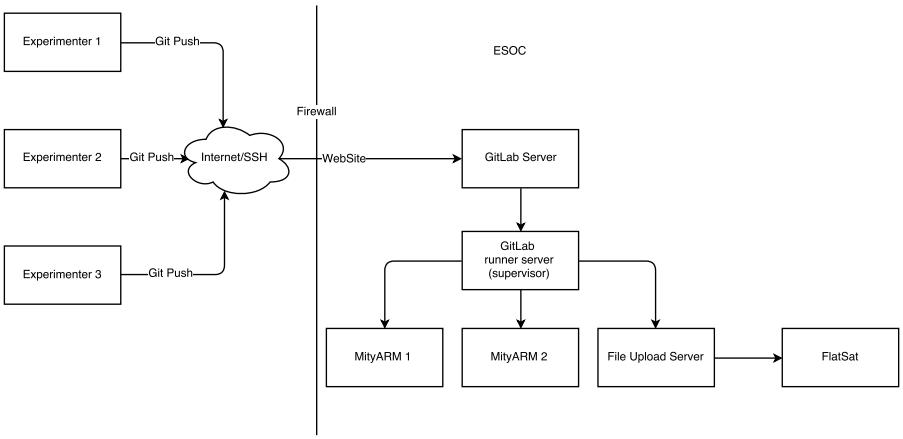


Figure 5.1.: Hardware architecture and interfaces

from running their own tests on the MityARMs. This would not be the case with parallel tests. The tags associated to the stages define which runner should pick up the job. There are two kinds of runners used in the architecture, shell and ssh. Shell means, that the defined test is executed on the GitLab runner server and ssh connects to a remote machine, executing the test there. The shell runner is configured to only run jobs that are tagged ”init”. This way, the initial tests can only be run on the GitLab runner server directly. The MityARM is registered as an ssh runner. It only runs tests tagged ”M1”, as seen in Listing 5.3.

5.2. Interfaces

For the registration of an experiment, an administrator creates a user for the experimenters. GitLab automatically sends the experimenters an email, that contains a temporary password and the uniform resource locator (URL) to the GitLab server at ESOC. At the first login, the experimenter must change the password and is then able to access the account that an administrator created for him. Then, the

experimenter has to create a public ssh key on their machine and enter it in their profile. GitLab provides a ”how to” in the ssh section, if an experimenter has never done this before. The experimenters are now ready to provide their first push to the project. The second interface for experimenters is the Website. As stated in Section 5.1, the website provides access to the users project. There they can see their account as well as the log and regression views.

The interface between GitLab server and supervisor is implemented following the GitLab installation process. In the GitLab runner setup, the URL for the corresponding GitLab server has to be entered, this way the runner knows where to listen for jobs.

The interface between the initial tests project defined by the administrators and the experimenters project is a trigger and a webhook. An example for a webhook is shown in Listing 5.1. When the experimenters do a git push command to the their project, the webhook is activated. The webhook is individual for every experiment and contains a variable with the link to the experimenter repository and the identifier (ID) of the experiment. The webhook also contains the token that connects to the trigger in the initial tests project. When the webhook is activated, so is the trigger with the corresponding token. This way, the initial tests project knows which experiment wants to be tested and pulls the data from the repository stated in the variable of the webhook. The tests then are executed in the experiment folder.

Listing 5.1: Webhook

```
http://URL/api/v3/projects/1/ref/master
/trigger/builds?token=[TOKEN]&variables[repoName]=
git@URL:NAMESPACE/Experiment1.git
&variables[expid]=1
```

The example for a webhook in Listing 5.1 shows the URL of the webhook. The first part of the webhook, until the first question mark, is the name space and address of the initial test project. The complete url is not shown due to security reasons. The token section includes the unique token of the trigger created in the initial test project, as described in Section 5.2. This connects the webhook to the trigger. The ”&variables[repoName]” section is the variable containing the link to the experimenter repository. This section is unique for every experiment. The last part is the variable that stores the experimenter ID. In a future implementation, there can also be variables for the experimenters email address, to automatically send emails in another stage of the ”.gitlab-ci.yml” file.

5.3. Testing procedure

First, the administrators at ESOC create a project with a script that defines the initial tests. This script contains the tests, such as checking if the filesize is within the proposed limits, or if an installation script is present. The script is also able to create ".txt" files containing the results of the test. Those files will be stored after the execution of the test script. The script is shown in Listing 5.2.

Listing 5.2: verify.sh script for initial tests

```
#!/bin/bash

echo "starting example CI verification script"

 # if [ -e *.sh ]; then
        #echo "Script is there, OK" >> result.txt
#else
#       echo "Script is not there, FAIL" >> result.txt
#       exit 1
#fi

du -sh >> result.txt
```

The script first echoes that it starts the example verification script. It then checks if the folder contains a script. This is just an example, for the actual tests, the verification script should look for the exact files and scripts. After that, an experimental printout of the file size is created. In the finished script, this should also be an "if" statement checking if the experiment exceeds a maximum file size.

Also, a ".gitlab-ci.yml" file has to be created in the initial tests project. This file is mandatory for the use of GitLab's continuous integration (CI) functions. Functions used for initial testing in this project are staging, artifacts and tags. The ".gitlab-ci.yml" file defines how the test will run. For example when the script will be executed, what happens if the test fails and how the artifacts are created. Only the administrators will have access to this project.

The experimenters have to provide an installation script for their experiment in their GitLab project, with the experimenters ID as name. On the satellite, the experiment will be installed by a script that installs the binary files that were uploaded together with the script, as mentioned in Section 2.2. This way, the procedure on the ground is the same as in flight. In the ".gitlab-ci.yml" file, the script has a variable name, which is called "runexp"$expid".sh", utilizing the unique ID of an experimenter as a variable called "expid". The variable is defined in the webhook of each individual project. Only when an experimenter pushes its experiment to the

server the webhook activates the trigger of the initial tests project and the variables from the specific experiment are transferred to the ".gitlab-ci.yml" file.

Listing 5.3: .gitlab-ci.yml file

```
before_script:
 - echo "Starting"
 - git pull "${repoName}"


stages:
 - initial_testing
 - experimenter_testing

initial_testing:
 stage: initial_testing
 script:
 - chmod u+x ./verify.sh
 - ls -l ./verify.sh
 - ./verify.sh
 tags:
 - init

 artifacts:
  when: always
  paths:
  - result.txt

experimenter_testing:
 stage: experimenter_testing
 script:
 - chmod u+x ./runexp"${expid}".sh
 - ls -l ./runexp"${expid}".sh
 - ./runexp"${expid}".sh
 tags:
 - M1
```

The example Listing 5.3 shows a possible implementation for testing experiments in the ".gitlab-ci.yml" file. The implementation has been tested and is documented in Chapter 6. This file, which is created and maintained by the administrators, defines how the test executes. It is divided into three subdivisions. "before_script" will always be executed before the rest of the stages, that are defined in "stages".

First, the git repository with the name of the experiment to test is pulled, so that the data is available. In GitLab, a stage can only start when the stage before it returned a positive test result. The criteria are defined by the administrators. This means, that first "initial_testing:" is executed and only when it returns a positive result the stage "experimenter_testing" is started. This ensures that an experiment can only be tested further if it passed the initial tests specified by ESOC. Examples for initial tests are: Checking if file size is less than a boundary or if a installation script is present to install the experiment. The tag "init" provides that only runners that are allowed to execute jobs tagged "init" can execute the job. The "artifacts" section determines what happens with files that are created in the test process. Here, the artifacts are always created, even when the test fails. This ensures, that the experimenters get notified why their tests failed. The path variable sets the path where to safe the file. The artifacts are stored and can be accessed via different methods as described on the GitLab website [6].

The stage "experimenter_testing:" again is defined to be executed in a shell. But this time only runners that are allowed to run tests with the tag "M1" are able to run the test. The experiment is then installed on the corresponding runner with the installation script with the unique experiments ID. This feature is now in place in theory and in the ".gitlab-ci.yml" file, but is not yet operational due to restriction issues on the MityARMs. Once the experiment has passed the initial tests, it is then able to run its own tests on the MityARM hardware. This is a security risk, for the experimenters as well as for ESOC.

For the experimenter it is risky, because the experiment would stay installed on the MityARM, enabling another experiment to see the experiment data. To solve this, it would be ideal if every experimenter would find the same starting conditions. To provide this feature, there would have to be other stages in the ".gitlab-ci.yml" that is created by the administrators. First to provide a the same starting conditions for every experiment and another one to cleanup after an experiment, erasing all files. This could be implemented by utilizing Docker. Docker is a software that is able to provide software containers with defined input and outputs. One such container could be created with an initial condition of the MityARM image. Docker then can call the image when an experiment is about to be tested on a MityARM and the image can be deleted after the test, with all the leftover experiment data. Reports and results created during the test would of course have to be extracted before deletion. This again could be done with the artifacts feature of GitLab.

The risk for ESOC is due to the fact that experiments can run scripts on the MityARM without security checks. This again could be solved by utilizing Docker. In the Docker image, that is to be used on the MityARM, the outputs could be restricted or even completely disabled. This way, the harm could be contained in the Image and deleted afterwards.

In the future, two more stages could be implemented. First, a stage for automatic email notification when the test is finished with a link to the created artifacts and secondly the stage for automatic file transfer to the flatsat. For automatically sending emails, for example sendmail could be configured, as it is a mail agent already installed on the provided virtual linux machine. It would then be possible to insert a command to send an email automatically to the experimenters. Further information in chapter Chapter 7.

The stage for automatic file transfer would implement a section that automatically transfers the experiments files onto the flatsat and installs them. This would only be possible when the initial tests passed and the experimenters are satisfied with their own tests on the flatsat. A scheduling mechanism for the experiments on the flatsat is required before implementing this stage.

A wiki function is also already in place, providing the experimenters with general data about the project, and ”how to” information. The wiki is located in its own project, and experimenters er granted access to it, but only with reading permission. This way it is ensured, that only administrators can alter the wiki. For each experiment, the wiki function is also in place. This way, experimenters can also provide the administrators with information about their software easily.

6. Results

This chapter recapitulates the assigned task and shows how the final realization of the architecture was implemented. There is also a section that documents the results in form of screenshots, that were created during a validation run. This run shows the interfaces for experimenters, for administrators, and shows the test process for an example git push command.

6.1. Recapitulation of Assigned Task

The task stated by ESA and esa was to create a test and control framework for the OPS-SAT project. As a requirement it should include the following entities: Experiment developers, satellite operators, offline test suits, log and regression view, communication and a runtime flight system.

The architecture introduced in section 4 identifies possible error sources in the lifespan of a satellite, and suggests several solutions for error avoidance. For example common hardware errors as well as software and operational errors and their according avoidance options. This is the first part of the test and control framework. Together with focus on the errors caused by experimenters they are the specification part of the test and control framework. The implementation that was proposed in Chapter 5 is the practical implementation of the test and control framework, and Section 6.2 shows the results.

6.2. Realization of the Architecture

The actual realization of the GitLab server and the GitLab runner server is as virtual machines on the ESOC server architecture. This way, it is easy to erase, or to change something. This could come in handy, if the current disk space of 20 GB is not sufficient enough anymore for the experiments. About 5 GB are used by the operating system and GitLab, which calculates to 15 GB of free space. With 115 experiments registered so far, the space for one experiment calculates to about 130 MB.

The single ground station of OPS-SAT will be located at ESOC in Darmstadt. With the planned orbit, the visibility time of the satellite is about eight minutes. The satellite is capable of a data rate of 200 kbit/s in the 256 kbit/s uplink rate. According to CCSDS standards, the data has to be Reed-Solomon encoded and interleaved, to minimize data loss in the propagation channel. This then results in a maximum data rate of 173 kbit/s for experiments according to [9]. Eight minutes, or 480 seconds, of data transfer with a data rate of 173 kbit/s leads to a file size of maximal 83040 bits or 10.380 bytes, which is about 10 Mb per experiment. Thus the 130 MB planned for every one of the 115 first registered experiments is sufficient.

No experiment is in a stage, where it could provide ESOC with data about file sizes. Thus having a virtual server that is able to grow is useful. The GitLab server uses one core of ESOC architecture's CPUs and has 2 GB of RAM associated, both values can also be adjusted to better fit the future needs. GitLab itself is able to host as many projects as there is free memory available. This is perfect, as it is not sure yet how many experiments will be registered in the future. The GitLab runner server has the same specifications. It has to store the experiments as well as the created artifacts. The artifacts can be automatically deleted after a time has passed. At this moment it is unclear in what rate experiments will have to be tested and thus how many artifacts have to be stored. The flexibility of the server comes in very handy in this case.

Running the initial tests will take about two seconds, with only very simple tests in place right now. The team is currently deciding on possible initial tests (Date: 09/09/2017).

6.3. Validation Run of Testing an Experiment

This section will show the results that were created during a validation run of the experiment testing architecture.

Image Figure 6.1 shows the overview of experiments so far, from the view of an administrator. Experimenters will only see their own project.

Some details in the screenshots, that would show ESOC internal features or details about specific information, are censored due to security concerns.

On top is the project that hosts the initial tests as described in Section 5.3. The second project is the experimenters wiki project. This project houses the wiki pages that are available for all the experimenters, also mentioned in Section 5.3. The third project is a test project created by the administrator. This is mainly to test webhooks and using git push commands to triggering the test process. The fourth project is the initial test project again, but with another namespace, to test if a change is possible. On the right side the red crosses indicate the failed tests, caused by git pushes to the initial test projects. This is due to the fact, that the test stage

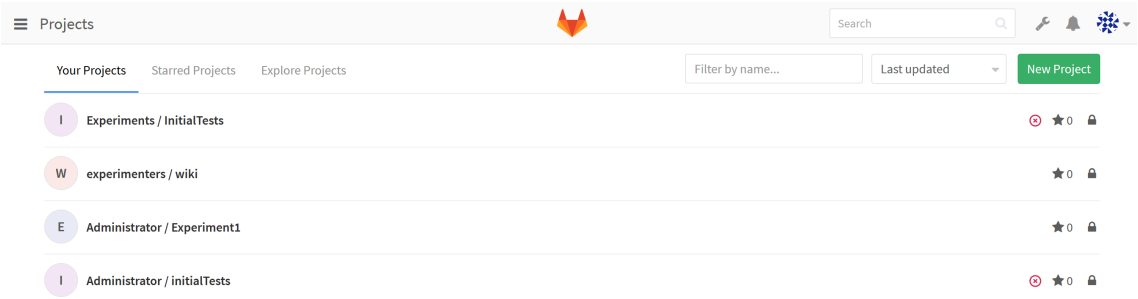


Figure 6.1.: OPS-SAT Overview of Experiments

that shall run on the MityARMs in a later stage is failing.

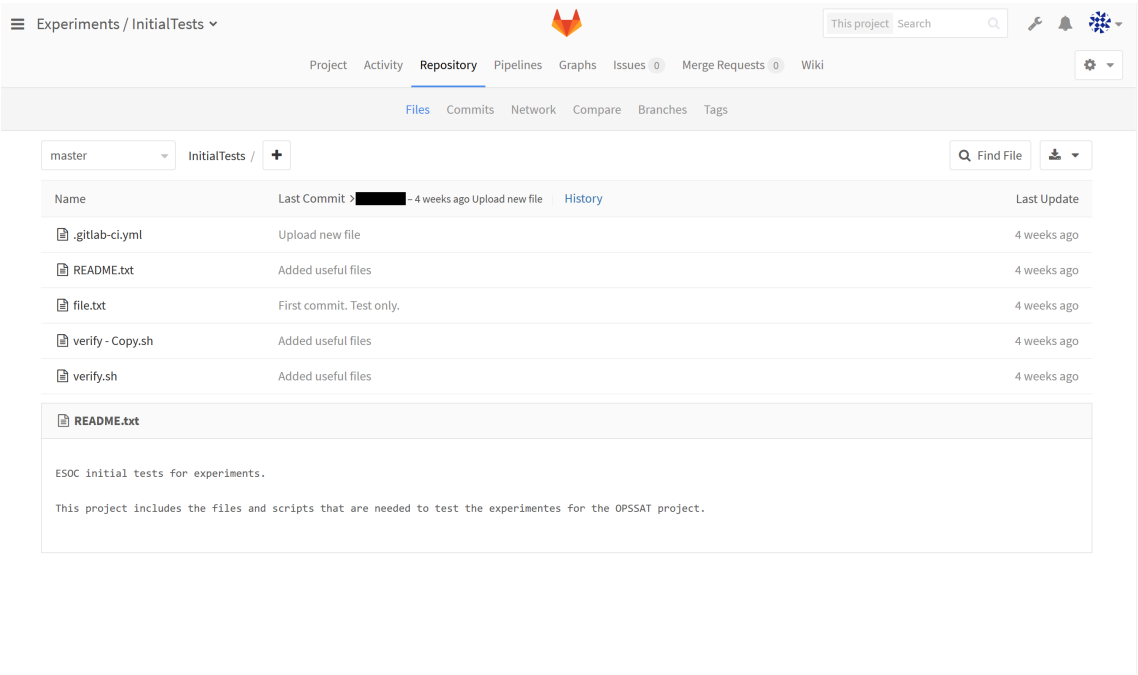


Figure 6.2.: View of Initial Test Project (Censored for Security)

Figure 6.2 shows a view inside the initial tests project. The top of the page shows the tabs associated with the project. The lower part of Figure 6.2 shows the repository tab, where the files with their last commits are shown. This contains the verify.sh script and the ".gitlab-ci.yml" file described in Listing 5.3. Also, a readme and a testfile are shown in the picture. The tab "commits" shows the history of changes, for the log and regression view. By clicking on the gear located on the top right of this page, the project options will drop down. This is where the options for triggers and runners are. The runners were created before hand according to

Chapter A.

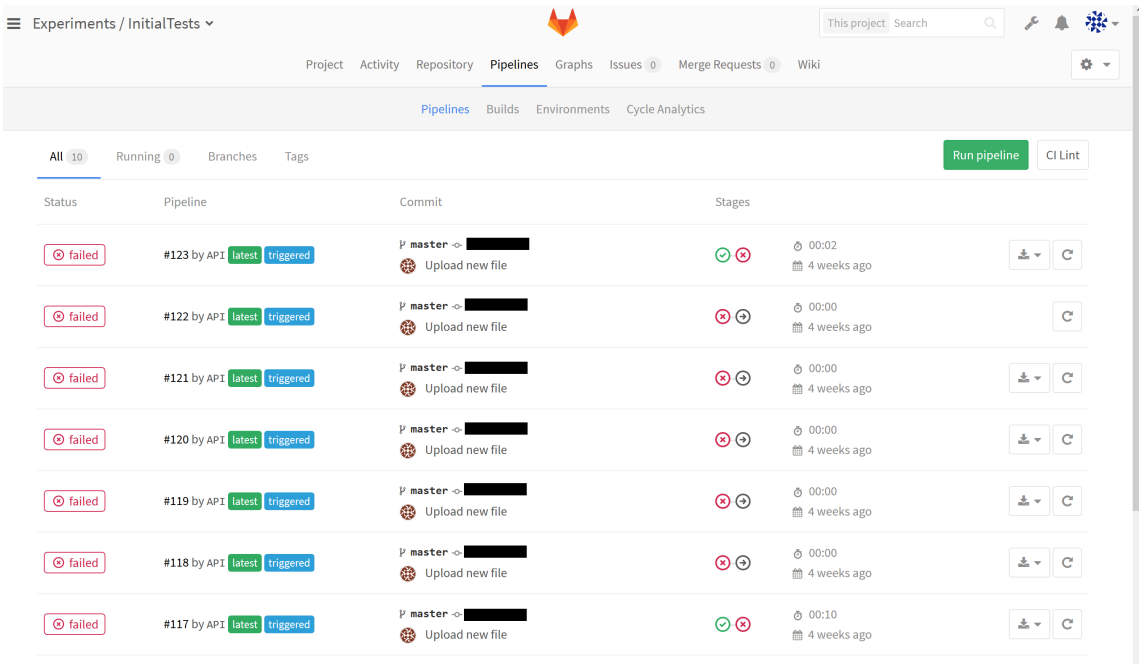


Figure 6.3.: Triggered Pipelines in InitTest project

The Figure 6.3 shows the pipelines that were triggered by git push events. The top of the page shows the interaction tabs, to navigate on the Website. This again is the view of an administrator. The experimenter will not see this, as they can only access their own project. When they push code to their own project, the test pipeline in the initial tests project is triggerd. This is an overview of all the events that caused the tests to be triggered. Now to inspect the first entry in more detail. On the left, there is a status icon showing if the test failed or succeed. Next is the job counter and who triggered it. The third column shows the commit of this push and a commit identifier, which is censored. The ”stages” column shows the two stages and their dependencies. First, the initial tests are executed, if they fail, like in the second row, the MityARM tests for experimenters is not allowed. In the first row, the initial test succeeded, but as described in Section 5.3 the MityARM stage is not yet available, thus the test fails. The fifth row shows the duration of the test and when it was triggered. The duration of the initial tests is between two and ten seconds, this is because of the very simple tests that are used in this example. On the far right, one can see a download button for the created artifacts and a retry button. As described in Section 5.3 the artifacts are always created. The ”verify.sh” script creates a report of the test, which is used to notify experimenters which part of the test failed or succeeded. The artifacts that are created in this

iteration of the project can not be downloaded by an experimenter. This feature has to be implemented in the future as described in Section 5.3 and in Chapter 7. The artifacts can also be downloaded via a link to the artifacts archive according to [6]. Using this feature in an automatically generated email will lead to further automate the test process. Further information in Chapter 7.

By clicking on the job, the administrators can see further details of the stages and if they succeeded or failed. Figure 6.4 shows this overview and the associated job. This is the detailed view of job 123, and it shows again, that the initial tests executed with a positive result and the experimenter testing failed, because of the missing MityARM.

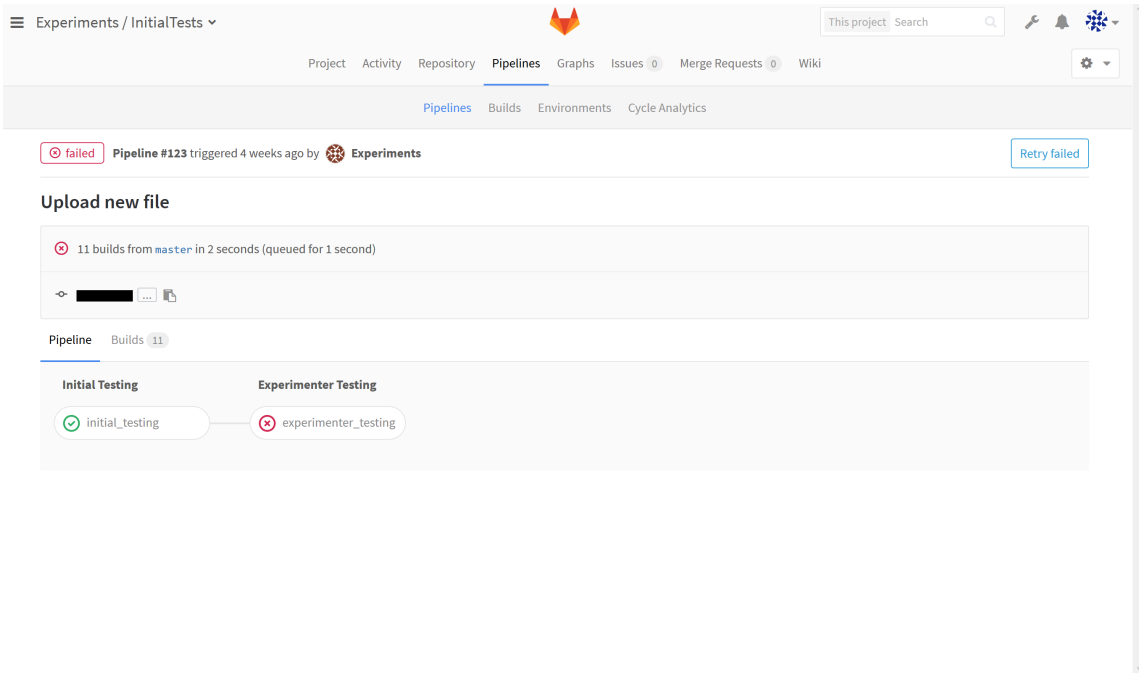


Figure 6.4.: Detail View of one Job

With another click on the stage, it is even possible to see an exact view of how the shell that executed the initial tests on the GitLab runner server. On top of the page is again the information about the build and the pipeline. The console shows the version of GitLab runner that is installed on the GitLab runner server (OSRUNNER). The testprogram outputs that it uses shell execution and runs on the GitLab runner server as mentioned in Section 5.1. This is due to the Tag ”init” that is visible on the right hand side. The next output the console shows is, that it clones the repository, and starts executing the ”.gitlab-ci.yml” file of the initial tests project. The detailed ”.gitlab-ci.yml” file can be found in Listing 5.3. The git config lines are from an older version of the ”.gitlab-ci.yml” file, where there had to be login to a specific git user. Then the console echoes the variable of the experimenters repository, in two different ways for testing purposes. Then the actual experiment is cloned and the ”verify.sh” is executed. At the end of the console, the log for the artifacts can be seen. This takes the files that the ”verify.sh” script created and saves them in the artifacts archive. In the future, it will be possible to send a link to the exact artifacts created by the test automatically as mentioned in Chapter 5 and in Chapter 7. For this, the unique token is used that is created in this step on the bottom of the page. It will be accessible as a job specific variable after the creation of the artifacts in the stages.

On the top right, the artifacts are ready to download, but only for the administrators thus far.

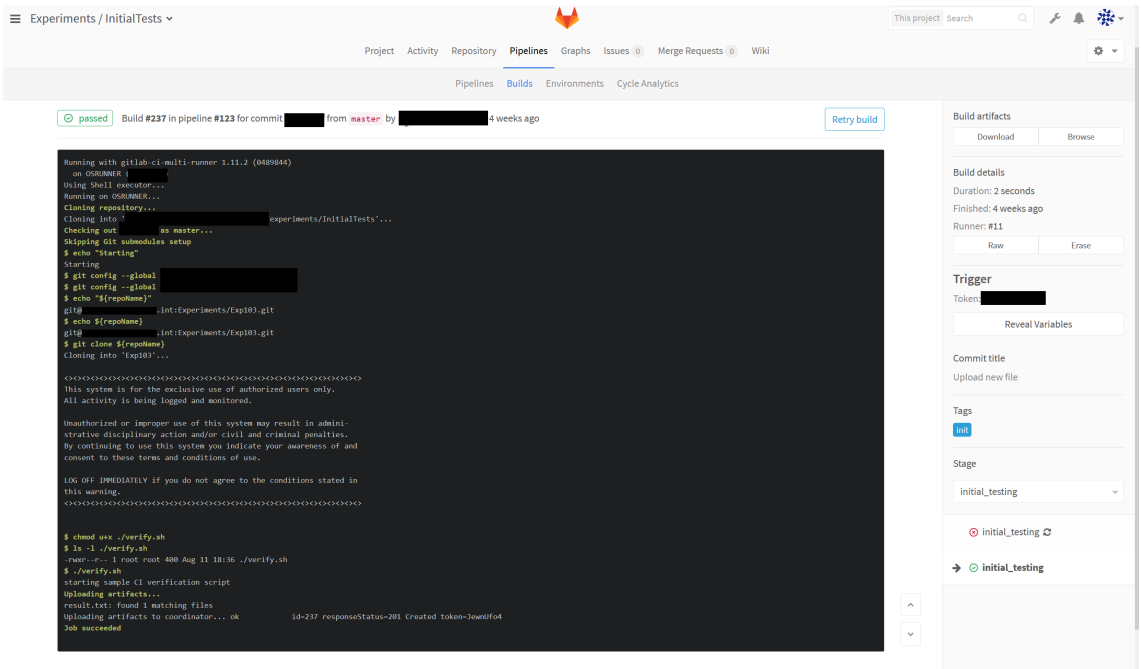


Figure 6.5.: Shell View of Job

Now to have a look inside a project that was created by an administrator for an experimenter.

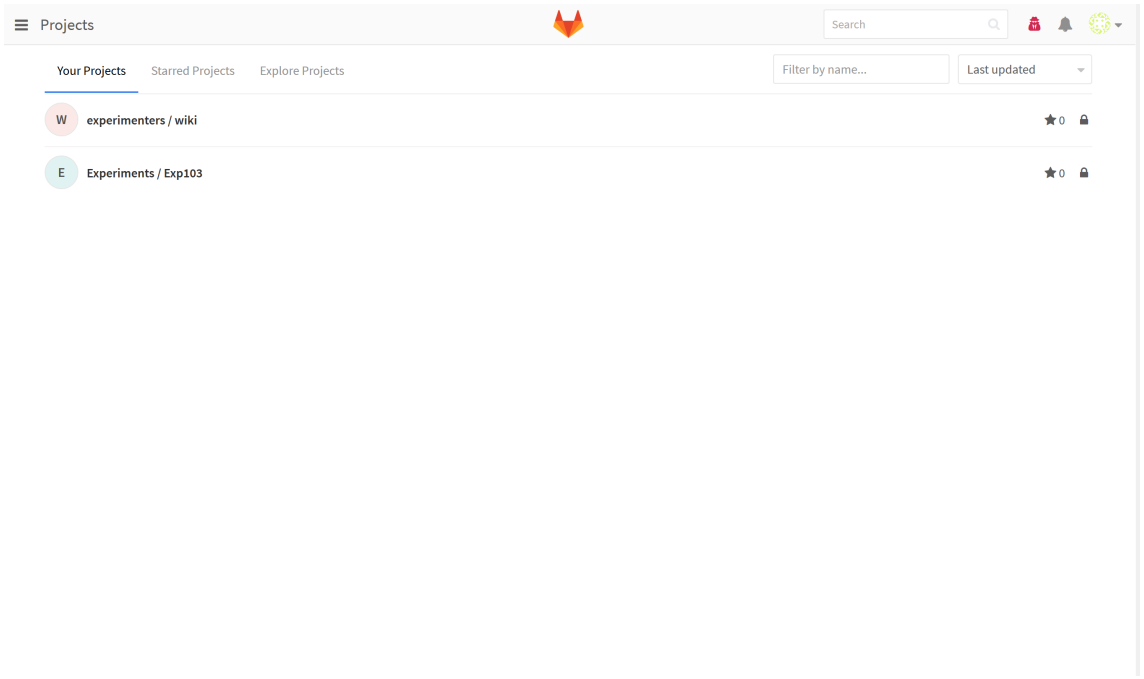


Figure 6.6.: Project Overview Experimenter

In Figure 6.6 one can see the experimenters overview of projects. The experimenter only sees its own project and has access to the wiki project for general information. On the top right the red symbol indicates the impersonation of an experimenter, which is possible to do as an administrator, and was used for demonstration purposes. The impersonated experimenter is also not a real experimenter, it is only a test user.

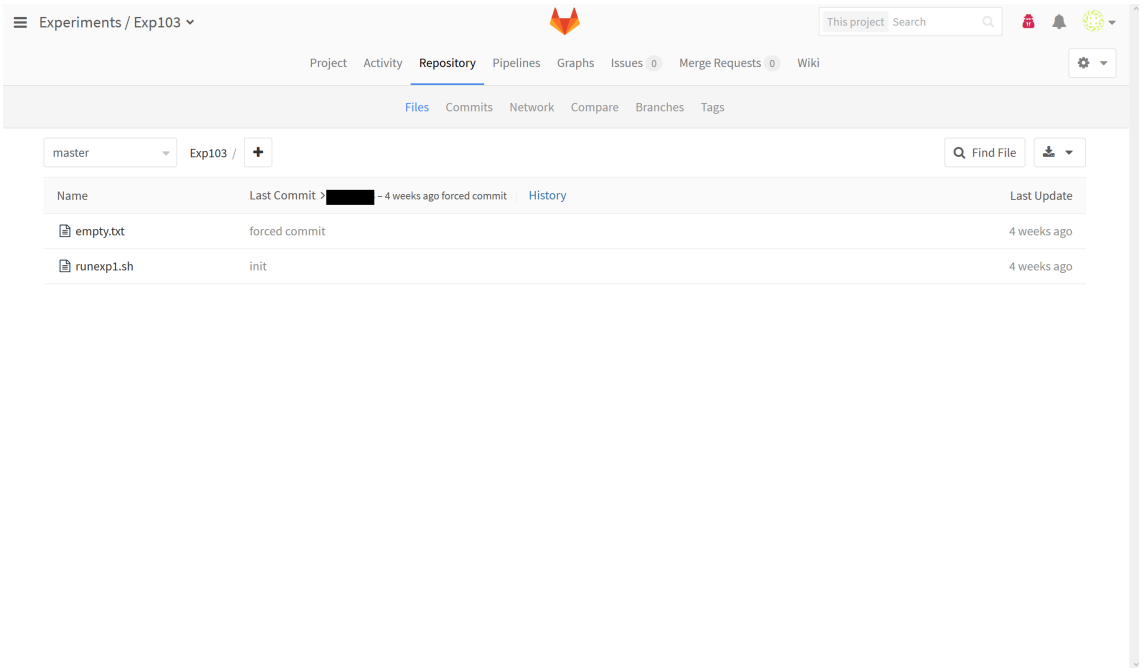


Figure 6.7.: Inside an Experiment with Experimenter View(Censored for Security)

Figure 6.7 shows the view inside an experiment with an experimenters view. One can see that the script is correctly inserted and that no binary is uploaded yet. In the stage that the project is now in, with the simple test methods in the initial test project, this leads to the experiment passing the initial tests. The button ”history” will the log and regression view of the project. This is visible for experimenters and administrators.

7. Conclusion

7.1. Summary

The current state of the architecture implementation is sufficient for registering experimenters to projects and allow them to push their own experiment data in form of binaries and an installation script to the project. There, a log and regression view is created by the git implementation. With the git push command, the initial tests are triggered and a report is created. The report is saved as an artifact and visible and downloadable for the administrators. The ”.gitlab-ci.yml” file of the initial test project has a second stage implemented to execute the experiment on a MityARM, but the MityARM is not yet correctly configured. The experimenters also have access to a wiki created by the administrators. In the future, there could be many more features implemented. The following section will show a few of them and how to implement them into the existing architecture.

7.2. Future Work

7.2.1. Email Notifications

Notifying experimenters about the outcome of their experiments is a crucial feature of the architecture, that has not been implemented yet. For this feature to work, first, one has to setup ”sendmail” , to be able to send emails automatically and by utilizing shell commands. An simple mail transfer protocol (SMTP) server is suffiecient, because the server will only send emails. With a simple SMTP server it will then be possible to enter a command into the ”.gitlab-ci.yml” file to enable automatic emails. The webhook associated with every experiment has to be changed, so that it includes the experimenters email address as a variable as mentined in Listing 7.1. This way it is easy to implement the automatic email transfer as a stage in the ”.gitlab-ci.yml” file of the initial test project. Listing 7.1 shows a possible webhook, again not with the actual url of the GitLab server at ESOC due to security precautions. It is the same webhook as in Listing 5.1, only with another example variable for the experimenters email address.

Listing 7.1: Webhook with email adress

```
http://URL/api/v3/projects/1/ref/master
/trigger/builds?token=[TOKEN]&variables[repoName]=
git@URL:NAMESPACE/Experiment1.git
&variables[expid]=1&variables[email]=user@domain.com
```

The webhook triggers the execution of the ".gitlab-ci.yml" file created by the administrators to test the experiment, just like in Section 5.2 of Chapter 5. The file has to be changed as well, to send emails automatically.

Listing 7.2: Example .gitlab-ci.yml File With Email Notification

```
after_script:
 - echo "You can find your test results at
  https://example.com/<namespace>/<project>/-/jobs/artifacts
  /<ref>/raw/<path_to_file>?job=<job_name>"
  | mail -s "Automated Test Results" "${email}"
```

By extending the same ".gitlab-ci.yml" file as in Listing 5.3 with the example code in Listing 7.2 the programm sends an email with the link to the created artifacts. The code shows the "echo" function that sends an email with the subject "Automated Test Results" to the mail address specified in the experiments unique webhook as shown in Listing 7.1. The body of the mail contains a description and the link to the artifacts that were created in this test. In this example for linking artifacts from [6] the parts wrapped in "⟨⟩" are placeholders for its content.

7.2.2. Testing on MityARM

The experimenter testing their experiments on the MityARM, as described in Section 5.3 is not yet operational. Yet it is necessary for the experimenters to have the option to test their own experiments on ESOC hardware. The stage in the ".gitlab-ci.yml" file is inserted and the MityARM registered as a runner. Testing on the MityARM in this stage of the project would be a security risk for the experimenters as well as for ESOC. As mentioned in Section 5.3 Docker could be used to create an image of the MityARM that provides only necessary, and by the administrators defined input and output options to the experiment running on the MityARM. This way, every experiment can have a clean image at the beginning of the test, without completely resetting the MityARMs operating system with every new experiment

test. The experiments would have access to the peripherals defined in the input and output section, just like with a regular MityARM. After the execution of the test, the created results can be stored in artifacts, and the image with the remaining experiment data deleted. This way, the damage potential of an experiment and the potential for experiments spying on each other is minimized. GitLab has a function to have Docker images as runners already installed in its latest runner verions. This way, only an update of the GitLab runner server would be needed.

7.2.3. Automatic Data Transfer Server

The test and control framework so far is limited to the initial tests. In the future, before uploading the experiment, it would also be desirable to test the whole process of uploading, installing and running the experiment on the ground. For this purpose the engineering model or "flatsat" is built. This final test can so far only be performed by ESOC team members, that do all the data transfer from the file upload server to the flatsat by hand. An automatic process would be desirable due to the low budget nature of the OPS-SAT mission.

In the future, there could be another server created and configured as a GitLab runner to schedule the experiments for final testing on the flatsat and carry out the data transfer from the server, through the data uplink and through the propagation channel simulator onto the flatsat. This way, the OPS-SAT team would only be needed to simulate the TMTC commands for the experiments, together with the experimenters.

7.2.4. Runtime Flight System

To further minimize the error sources, performance logs could be created during the phase were experimenters test their experiments on ESOC hardware, such as MityARMs or the flatsat. This way, the administrators would be able to collect data about the experiment and derive boundaries from it, for when the experiment is executed in orbit. Resources to be logged could be CPU usage, RAM usage, Battery usage data rates and so on over time. On the satellite, a software could be installed that monitors those resources and at the same time compares them to the created logs to give warnings when a process uses too much processing power too long. This way, not only the operators get an early warning if something is not exactly right with an experiment, also the experimenters could implement functions that recognize a boundary breach event, to stop the affected software part. If that still does not help, the experimenters can implement a software function that stores the data created until that time, to recover part of the created results.

7.2.5. Battery Model

In the more distant future, it would also be possible to create variable power source for the flatsat, in order to simulate the power production by the solar panels. The simulation could be based on a model of how the solar panels produce power in the different orbit sections. The model could be precise enough to plan the available power for the experiment on the day it is executed. When the experiment influences the satellites attitude in a deterministic way, even the power production with the satellites precise attitude with orientation to the sun could be calculated. This way, the power consumption of an experiment can be monitored precisely. And as an advantage for experimenters, they can further optimize their operations.

A. Installation Process

1. Get a copy of the virtual machine, provided by Thorsten Graber in SMILE lab. Make sure to have root access to the server.

2. Download Mobaexterm on your machine to have a way to SSH into the server. Start a new session and change the user to root with the command su root. Enter the password.

3. Initialise the machine for the installation of GitLab with the information provided on

   ```
   https://about.GitLab.com/installation/#opensuse
   ```

4. Execute step 1

5. For the second step, script installation is not possible. You must download the files manually and run them.

6. Download a version of GitLab from

   ```
   https://packages.gitlab.com/gitlab/gitlab-ce
   ```

   For the file, search for "opensuse/" as distribution and download the latest version of the install files for sles13.x86.

7. Right click on that and copy link address

8. Download the file using

   ```
   cURL: curl -LJO https://packages.gitlab.com
   /gitlab/GitLab-ce/packages/
   opensuse/13.2/gitlab-ce-8.14.0-ce.0.sles13.x86\_64.rpm
   /download
   ```

9. Install the package with:

   ```
   rpm -i gitlab-ce-8.14.0-ce.0.sles13.x86\_64.rpm
   ```

10. Reconfigure GitLab with:

    ```
    sudo gitlab-ctl reconfigure
    ```

11. Change the external URL of the Server to its IP, so that it is reachable.

12. The config file is reachable with:

    ```
    sudo vi /etc/gitlab/gitlab.rb
    ```

13. Using the arrow keys navigate to the parameter external URL and change it by pushing, which lets VI get into insert mode.

14. To end VI press escape to go to the normal mode, then type ":wq" to save and quit the program. If you made a mistake, end the program without saving by typing "q!" in normal mode.

15. Reconfigure GitLab using:

    ```
    sudo gitlab-ctl reconfigure
    ```

16. Enter the IP-adress of your server in your webbrowser

17. Enter the password you desire

18. Log in with username: root and the password you just created

19. To push projects, you have to get a SSH key.

20. Go to "profile settings" in the top right corner and then to SSH Keys.

21. Follow the instruction on creating a SSH key.

22. Insert the key.

23. Create a new project by manoeuvring to the top left corner, open the menu and select "projects"

24. create the projects for the initial tests

25. add a trigger in the project options

26. IMPORTANT: Copy the example for Webhook shown on the bottom of the page

27. safe the project

    Install GitLab Runner on the GitLab runner server

28. GitLab runner and GitLab server shall not run on the same machine, because the runner can, by using scripts, alter the server.

29. The Version of GitLab that is available for the SLES version 11 is 8.14.0. For this version, the runner has to be version 1.11.2, because runner version 9.xx cannot connect to GitLab 8.xx, it needs GitLab version 9.xx.

30. Download the binary with command:

    ```
    wget –O /usr/local/bin/gitlab-ci-multi-runner wget –O /usr/local/bin/
    ```

31. Give executable permission with

    ```
    chmod +x /usr/local/bin/gitlab-ci-multi-runner
    ```

32. Register a runner using:

    ```
    gitlab-ci-multi-runner register
    ```

33. Put in the address of the coordinator, in our case osgitlab.esoc.esa.int

34. Get the token for the runner from the admin area, then go to runners

35. Insert the token

36. Write the description

37. Write the tags (for the initial tests runner its ”init” this ensures that the runner only runs the initial testing jobs. When you configure a MityARM as a runner, it will get another tag, so it only runs scripts like it would on the satellite)

38. do not let it run untagged builds

39. for the init tests select shell as executor

40. after successfully registering the runner, install the program with:

    ```
    gitlab-ci-multi-runner install --user=root --working-directory=/usr/l
    ```

41. run the service with: gitlab-ci-multi-runner run

42. start the service with: gitlab-ci-multi-runner start

43. check if the service is running with: gitlab-ci-multi-runner status

44. check if the GitLab runner server has a SSH Key for pulling the repositories from the GitLab server, as user admin.

45. Type:

    ```
    cat ~/.ssh/id_rsa.pub
    ```

    in the command line, if the machine has a public key, it will be shown here. Copy the whole text.

46. Go to the GitLab instance as admin, on the top right, select profile settings on your Admin logo. There, move to SSH-Keys, Insert the key and give it a description.

47. If there was no public SSH key in that location, create one with the manual provided on the admin page.

    Register new experiment

48. go to admin area – users – create user

49. insert email address and name of user

50. the user has to be marked external

51. create user

52. go to projects – new project

53. name the project with its unique ID

54. add the created user as developer or master

55. register the new project

56. go to the new project

57. add a webhook following the instructions on the copied example before

58. add the Experimenter repository as a variable using

    ```
    &variables[repoName]=ACTUAL LINK
    ```

59. add the ID using

    ```
    &variables[expid]=ID
    ```

    Testing the project

60. after an initial push, the webhook can be tested. If there is no push, the webhook can not be triggered manually

B. Flowchart for Experiment Test

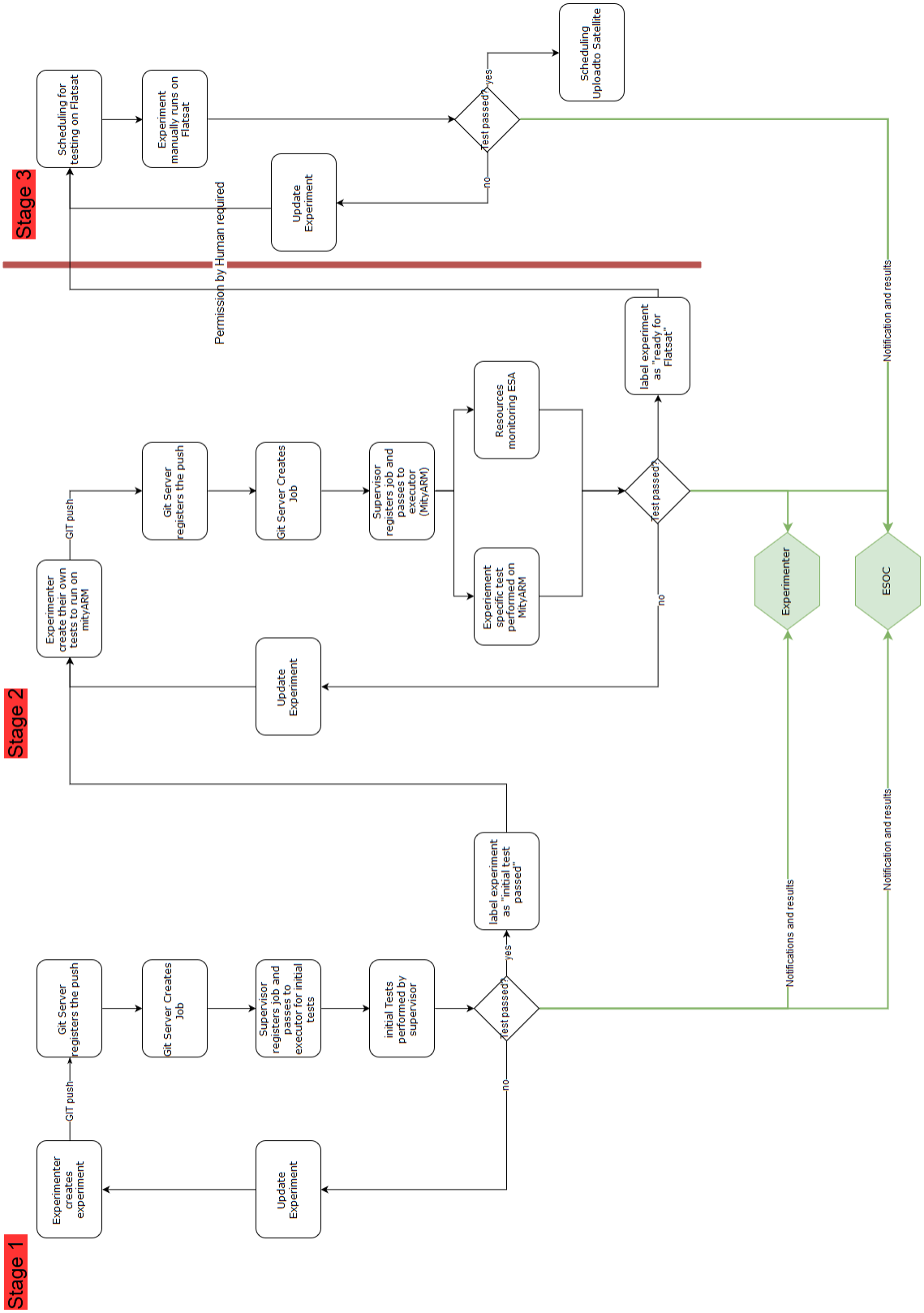


Figure B.1.: Overview of Experiment Testing Process

List of Figures

2.1. OPS-SAT2NM 18

4.1. Experiment architecture overview 27
4.2. Summary Evaluation1 29
4.3. Summary Evaluation2 30
4.4. First Stage of Experiment Testing Process 36
4.5. Second Stage of Experiment Testing Process 37
4.6. Third Stage of Experiment Testing Process 38

5.1. Hardware architecture and interfaces 40

6.1. Project Overview 49
6.2. Project with Initial Tests 49
6.3. Triggered Pipelines in InitTest project 50
6.4. Detail View of one Job 52
6.5. Shell View of Job 53
6.6. Project Overview Experimenter 54
6.7. Inside an Experiment with Experimenter view 55

B.1. Overview of Experiment Testing Process VI

List of Tables

2.1. Radiation related failures of electronics in space from [7] 14

List of Listings

5.1. Webhook 41
5.2. verify.sh script for initial tests 42
5.3. .gitlab-ci.yml file 43

7.1. Webhook with email adress 58
7.2. Example .gitlab-ci.yml File With Email Notification 58

List of Acronyms

ADCS attitude determination and control system

ALM arithmetic logic modules

API application programming interface

CCSDS consultative committee for space data systems

CI continuous integration

COTS commercially available off the shelf

CPU central processing unit

EPS electrical power system

esa Embedded Systems and Applications Group

ESA European Space Agency

ESOC European Space Operations Center

FDIR fault detection, isolation and recovery

FPGA field-programmable gate array

GUI graphical user interface

HPS hard processing platform

ID identifier

JVM Java virtual machine

LAB logic array block

MLAB memory logic array block

NASA National Aeronautics and Space Administration

OBC on-board computer

OBDH on-board data handling

OPS-SAT Operations-Satellite

OS operating system

PC personal computer

RAM random access memory

SEPP satellite experimental processing platform

SMTP simple mail transfer protocol

ST8 Space Technology 8

TDRS tracking and data relay satellite

TMTC telemetry and telecommand

UHF ultra-high frequency

URL uniform resource locator

VHDL very high speed integrated circuit hardware description language

Bibliography

[1] H. Abakians, M. Bothwell, A. B. Chmielewski, R. M. Nelson, C. M. Stevens, J. Ku, M. E. McEachen, S. White, J. R. Samson, J. Zsoldos, and T. McDermott. "NASAs New Millennium ST-8 Project". In: *AGU Fall Meeting Abstracts* (Dec. 2006).

[2] *Ariane 501 - Presentation of Inquiry Board report*. Timestamp: 10/09/2017 17:14. URL: `http://www.esa.int/For\_Media/Press\_Releases/Ariane\_501\_-\_Presentation\_of\_Inquiry\_Board\_report`.

[3] M. M. Day. "30 Years of Commercial Components In Space: Selection Techniques Without Formal Qualification". In: *13th Annual AIAA/USU Conference on Small Satellites* (1999).

[4] *Description of Software Bugs*. Timestamp:10/09/2017 18:17. URL: `https://en.wikipedia.org/wiki/Software_bug#Typographical_errors`.

[5] D. Evans. "The Ladybird Guide to Spacecraft Operations". The Ladybird Guide to Spacecraft Operations is a presentation created by David Evans, that contains first hand information about possible error sources on satellite missions with examples. 2016.

[6] *How to download GitLab artifacts*. Timestamp: 02/09/2017 10:14. URL: `https://docs.gitlab.com/ee/api/jobs.html#download-the-artifacts-file`.

[7] A. Y. John Scarpulla. "What Could Go Wrong? The effects of Ionizing Radiation on Space Electronics". In: *Crosslink* (2003).

[8] M. Johnson. *NASA Ends Attempts to Fully Recover Kepler Spacecraft, Potential New Missions Considered*. Aug. 2013. URL: `https://www.nasa.gov/feature/ames/kepler/nasa-ends-attempts-to-fully-recover-kepler-spacecraft-potential-new-missions-considered`.

[9] A. T. Lange. *OPS-SAT System Summary for Experimenters - v1*. 1st ed. ESA/ESOC. Robert-Bosch-Strasse 5, 64283 Darmstadt, July 2015.

[10] *Mariner 1 mission description*. Timestamp:10/09/2017 18:02. URL: https://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1.

[11] *NanoSat MO Framework*. Timestamp: 09/09/2017 14:53. URL: https://en.wikipedia.org/wiki/NanoSat_MO_Framework.

[12] *OPS-SAT Architecture CDR v15-3_2Nanominds*. Critical Design Review for the OPS-SAT architecture.

[13] *OPS-SAT Mission Overview*. Timestamp: 07/09/2017 08:51. URL: http://www.esa.int/Our_Activities/Operations/OPS-SAT.

[14] D. K. Sahu. "EEE-INST-002: Instructions for EEE Parts Selection, Screening, Qualification, and Derating". In: *Nasa Publications* (2008).

[15] *SpaceX disaster 'looks like structural failure'*. Timestamp: 10/09/2017 18:41. URL: https://www.flightglobal.com/news/articles/spacex-disaster-looks-like-structural-failure-414818/.