Thomas Halvard Bolle

Operationalizing testing setup used at NTNU SmallSat Lab

Design of an Automatic Test-framework for On-board Software of Satellites

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
Co-supervisor: Roger Birkeland
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Thomas Halvard Bolle

Operationalizing testing setup used at NTNU SmallSat Lab

Design of an Automatic Test-framework for On-board Software of Satellites

Master's thesis in Electronic Systems Design
Supervisor: Milica Orlandic
Co-supervisor: Roger Birkeland
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

Satellites are complicated constructs that require interdisciplinary teamwork of various experts of different academic disciplines. At the SmallSat Lab at Norwegian University of Science and Technology (NTNU), a team is working on designing and developing satellites known as HYPSO (HYPer-spectral Smallsat for ocean Observation). The HYPSO satellites consist of a payload integrated with a 6U CubeSat satellite bus developed by NanoAvionics. With the satellites, the HYPSO team aim to use a hyperspectral imager to observe ocean color and detect harmful algal blooms (HABs) that can threaten ecosystems in seas and lakes. To ensure proper functionality of the satellites, the codebase for the on-board software require rigorous testing. Creating and executing these tests is a time-consuming process, that require intricate knowledge of the inner workings of the codebase. Additionally, as the codebase increases with new functionality added to the satellite, previous tests might become outdated. Which leads to more time being spent on maintenance.

This thesis describes the design and implementation of an automatic test framework for the on-board software of the HYPSO satellites. A test framework is a set of guidelines/rules for creating and designing test cases, while an automatic test framework is a test framework that is able to test automatically at certain events. Utilizing an automatic test framework will improve the HYPSO team's test efficiency, test accuracy and test maintenance cost.

The automatic test framework designed in this thesis is able to imitate a normal user's usage of the satellite and is able to efficiently test over multiple test cases, with multiple sets of input data. It has been implemented to execute testing automatically when changes are made to the codebase. As each test case is also timed and dated, possibilities arise for documenting/logging changes in performance speed of the on-board software. Which is useful considering the limited connection time the satellite has to a ground station.

Sammendrag

Satellitter er kompliserte konstruksjoner som krever tverrfaglig kompetanse og sammarbeid mellom personer mer ulike akademiske disipliner. Ved SmallSat Labben på Norges teknisk-naturvitenskapelige universitet (NTNU), jobber det ett team som designer og utvikler satellitter ved navnet HYPSO (HYPer-spectral Smallsat for ocean Observation). HYPSO-satellittene består av en nyttelast integrert med en 6U CubeSat satellite-bus utviklet av NanoAvionics.

Nyttelasten inneholder ett hyperspektralt kamera som HYPSO teamet vil bruke for å ta hyperspektrale bilder av havet for å kunne se etter tegn av alge oppblomstring. Som kan true økosystemer i både hav og innsjøer. For å forsikre at satellitten fungerer om det skal, kreves det grundig testing av kodebasen for programvaren ombord. Å lage disse testene er en tidkrevende prosess, som krever stor kunnskap til den indre virkemåten av kodebasen. I tillegg, så etterhvert som kodebasen blir større når ny funksjonalitet legges til, kan gamle tester slutte å fungere. Noe som da fører til at mye tid må brukes på vedlikehold av disse testene.

Denne oppgaven beskriver design og implementering av et automatisk testrammeverk for programvaren ombord HYPSO-satellittene. Et testrammeverk er et sett med retningslinjer/regler for hvordan en skal lage tester, mens et automatisk testrammeverk er et testrammeverk som er i stand til å kjøre testene automatisk. Å bruke et automatisk testrammeverk vil forbedre HYPSO teamets effektivitet, testnøyaktighet og vedlikeholdskostnader når det kommer til testing. Det automatiske testrammeverket designet i denne oppgaven er i stand til å imitere en normal brukers bruk av satellitten og er i stand til å effektivt teste flere tester om gangen, med ulike sett av ingangsdata. Testrammeverket er implementert til å automatisk utføre testing hver gang det skjer endringer i kodebasen. Og ettersom hver tests blir timet og datert, så gir det muligheter for å dokumentere/logge endringer i programvarehastigheten til satelliten over tid. Noe som kommer godt med, med tanke på den begrensede tiden man har for kommunikasjon når satelliten er over bakkestasjonen.

Acknowledgements

I want to thank all the members of the HYPSO team for all the support and help I have gotten throughout the semester. Especially my co-supervisor Roger Birkeland, who helped me with the decision-making when choosing the framework to implement. I would also like to thank Sivert Bakken, who helped me understand how GitHub Actions functions, and why it is such an amazing tool to utilize.

Lastly I want to thank my fellow students, Einar Avdem, Edvard Birkeland, Eivind Bjørnebøle, Simen Netteland and Anders Brørvik, who have made my last month's on NTNU be a time filled with joy.

Table of Contents

List of Figures

List of Tables

Acronyms

ADCS Attitude and Determination Control System. 5

CAN Controller Area Network. xi, 7, 8, 9, 11, 14, 33

CI/CD continuous integration and continuous delivery. 3, 25, 29, 63, 64

CLAW Colored Littoral Zone and Algea Watcher. 4

CLI Command Line Interface. 9, 10

CSP CubeSat Space Protocol. 7, 8, 9, 11

CSV Comma Separated Values. 36, 43

EPS Electrical Power System. 5, 14, 42, 43, 50, 55

ESD ElectroStatic Discharge. 14

FC Flight Computer. 5

GS Ground Station. 5, 7

HSI Hyper Spectral Image. 1, 2, 5, 7, 9, 10, 56, 57, 64

HYPSO HYPer-spectral Smallsat for ocean Observation. i, xi, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 23, 24, 26, 29, 30, 31, 32, 33, 42, 45, 46, 49, 50, 56, 61, 63, 64

JSON JavaScript Object Notation. 36

NTNU Norwegian University of Science and Technology. i, 1, 4, 5, 63

OPU On-board Processing Unit. 5, 9, 10, 33, 38, 42, 43, 49, 50, 56, 57, 64

OS Operating Software. 9

OSI Open Systems Interconnection. xiii, 6, 7

PC Payload Controller. 5, 8, 14, 56, 57

PL Programmable Logic. 6

PS Processing System. 6

REPL Read-eval-print-loop. 10

RGB Red-green-blue. 5, 7, 10

SDR Software Defined Radio. 1, 2, 4, 5, 14

SSH Secure Shell. 14

UHF Ultra-High Frequency radio. 5

UNIX UNiplexed Information Computing System. 40

USB Universal Serial Bus. 7

YAML YAML Ain't Markup Language. 25, 29, 46

1 Introduction

"Myth has become reality: Earth's gravity conquered"

French daily Le Figaro

When looking up at the sky, you can see many things including planets and start. But you can also see satellites. October 4, 1957, Sputnik 1, the first man-made satellite, was launched. On this day, as French daily Le Figaro said, Earth's gravity was conquered, and the start of the human space age had begun. At that time satellites were simple systems. The Sputnik 1 consisted of only four antennas and an aluminum sphere containing a low-power transmitter used for sending out signals that could be picked up on Earth [37]. But as major technology advancements have been made in the years since, so have the advancements in satellites. These days satellites are more complex than ever before, with multiple components that all need to work in tandem for the satellites to function properly. As satellites are "out of reach" as soon as they leave planet earth, it is imperative the on-board software function as expected. To this end, testing is an important process of the development, and facilitating this testing requires thorough and comprehensive tests. This thesis describes the design and implementation of an automatic test framework for the On-board software of the components in satellites, as a means to reduce the time cost and complexity of creating such tests.

1.1 The HYPSO Project

The HYPer-spectral Smallsat for ocean Observation (HYPSO) project is the first part of the Norwegian University of Science and Technology (NTNU) SmallSat Lab's initiative to grow space-related knowledge on NTNU. At the SmallSat Lab, the HYPSO team have designed and built the CubeSats used for the project. The first mission, HYPSO-1, focuses on bringing a hyperspectral imager into space using a CubeSat, with the aim of observing ocean color and detect harmful algal blooms through Hyper Spectral Images HSI, as seen in Figure 1 [50]. As the first mission is currently well underway, with the HYPSO-1 CubeSat soaring up in orbit, the team at the SmallSat Lab has started work on the second mission, HYPSO-2. The second mission is an improved version of HYPSO-1, with more features, specifically a second payload containing a Software Defined Radio (SDR) which shall be used for providing Arctic researchers easier and faster access to scientific data products regarding the Ultra-high Frequency (UHF) communication channel and on-orbit interference [51].

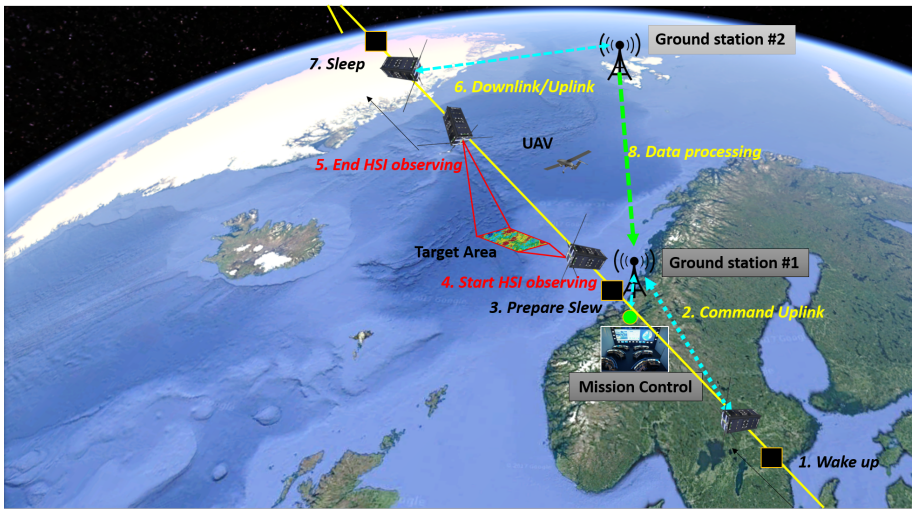
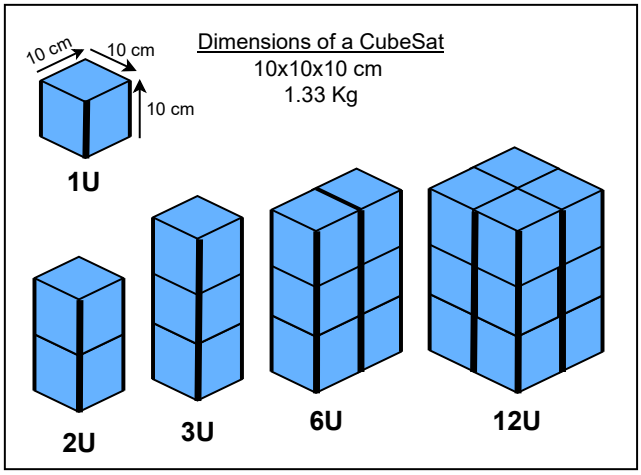


Figure 1: Image over HYPSO mission

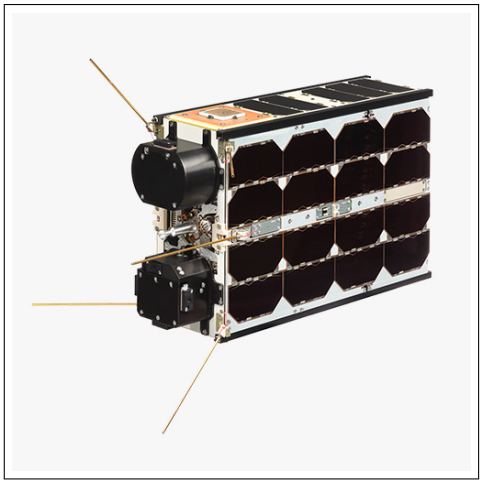
The HYPSO project is motivated by observing the effect climate change has on the ocean, specifically how climate change effects the growth of algal blooms that threaten the fish farming industry in Norway. These blooms may kill fish in various of ways, for example by depleting the oxygen in the water due to high respiration rate of the algae, or by bacterial respiration during their decay [30]. Detecting algal blooms early with satellites can help owners of the farms save their fish. Once the algal blooms are discovered, the satellite will downlink data to be a part of an Autonomous Ocean Sampling Network (AOSN) including Autonomous Surface Vehicles (ASV), Autonomous Underwater Vehicles (AUV) and Unmanned Aerial Vehicles (UAV), which can investigate the situation further. Using the HSI camera enables images to be taken across the entire spectrum from visible light to near-infrared light (400 nm-800 nm), as light is diffracted inside camera into separate wavelength. Hence, it is able to detect light given of algal blooms in the near-infrared area. Adding the SDR to this project for easier and quicker communication with the Arctic is important since it is one of the regions where consequences of global warming are observed early [8]. SDR's are flexible communication platforms that enable functional changes by modifying the software and keeping the same hardware. Which can be beneficial in a satellite for in-flight updates of channels and interference measurements, for communicating with autonomous vessels.

1.2 CubeSats

Both the HYPSO-1 and HYPSO-2 satellites are 6U CubeSat satellites. A CubeSat is a class of research spacecraft called nanosatellite, which are built up of cubic modules measuring 10 cm x 10 cm x 10 cm in size, with a maximum weight of 1.33 Kg per module [38]. The 6U CubeSat consist of 6 of these cubic modules, totaling 10 cm x 20 cm x 30 cm in size, Figure 2a. The exterior of the 6U CubeSat satellite can be seen in Figure 2b. The HYPSO CubeSats contain a M6P satellite bus developed by NanoAvionics and payload(s) created by the HYPSO team. NanoAvionics specializes in the production of satellite buses, and solutions for commercial and scientific missions for small satellites [3]. By having the satellite bus provided, the HYPSO team are able to concentrate on the payloads used to accomplish the mission goals of the CubeSats, and their support during the mission in orbit.



(a) CubeSat units.



(b) 6U CubeSat. Courtesy of NanoAvionics.

Figure 2: CubeSat dimensions.

1.3 Objective

To ensure proper functionality of the HYPSO CubeSats, the codebase for the on-board software requires rigorous and through testing. Tests should demonstrate that the implementation meets the functional and performance requirements of the CubeSats.

The HYPSO team consist of multiple PhD candidates and postdocs who stay on the team for long periods of time, as well as multiple Master's and Bachelor's students who join the team for specific projects spanning one or two semesters. Out of the team members, many are from multiple different departments and have different disciplines. In this setting the HYPSO team members usually have limited amount of time, and a high turnover. This coupled with the steep learning curve of the current test framework, which utilizes Jenkins for continuous integration and continuous delivery (CI/CD), creates a situation where very few tests are created and maintained. Due to this, most of the on-board software is not tested, and proper functionality of the HYPSO CubeSats can not be guaranteed.

The goal of this Master's thesis, is to analyze the current test framework, to find out why it is not utilized, and what possible improvements can be made. Before designing and implementing an improved Automated test framework. The new framework should contain an easier and more efficient way of creating and executing tests for the on-board software of the CubeSats, with a lower learning curve than the current version.

1.4 Structure of Master's thesis

The Introduction, Section 1 contains a brief introduction with a main focus on the HYPSO mission, the satellites, and the goal of the thesis. Followed by the background, which consist of two sections. Section 2, System background, contains background theory on HYPSO system to understand the work done. This includes explanations of the satellites, its structures and connections, and the test setup utilized by the HYPSO team. Section 3, Automated Test Frameworks, contain background information on types of testing, and different automation test frameworks. After the background sections comes Methods & Tools, Section 4. This section contains some of the tools and methods used by the HYPSO team, for developing the codebase for the satellites, and that are needed for understanding the work in this thesis. Then follows the Analysis section, Section 5, which contains an analysis of the current test framework used by HYPSO, and which of the automated test framework from Section 3, would work best for the new test framework developed in this thesis. Based on the analysis, a design is made, and then implemented in the Design & Implementation section, Section 6. After the new automated test framework has been implemented, tests are executed in the Result section, Section 7. Following the results, a discussion of the entire framework is presented in Section 8. This discussion is then concluded in, Section 9, Conclusion. Lastly Section 10, Future Work, contains work that can should be done for the utilization of the automated test framework, but that the author did not have time to do.

2 System Background

This Section contains background related to the HYPSO satellites, the test setup and the software used for this thesis.

2.1 HYPSO

As mentioned in the introduction, see Section 1, HYPSO team at the NTNU SmallSat Lab has designed and built one satellite, HYPSO-1, that is currently in orbit, and is already hard at work on the second satellite, HYPSO-2.

Figure 3 and Figure 4 illustrates the parts and interfaces between in the satellites. Both contain the identical M6P satellite bus (some of the subsystems in the M6P bus have been excluded from the figures) developed by NanoAvionics, and the Colored Littoral Zone and Algea Watcher (CLAW) payload developed by the HYPSO team. In addition, HYPSO-2 also contains a second payload which houses the SDR with the name TOTEM.

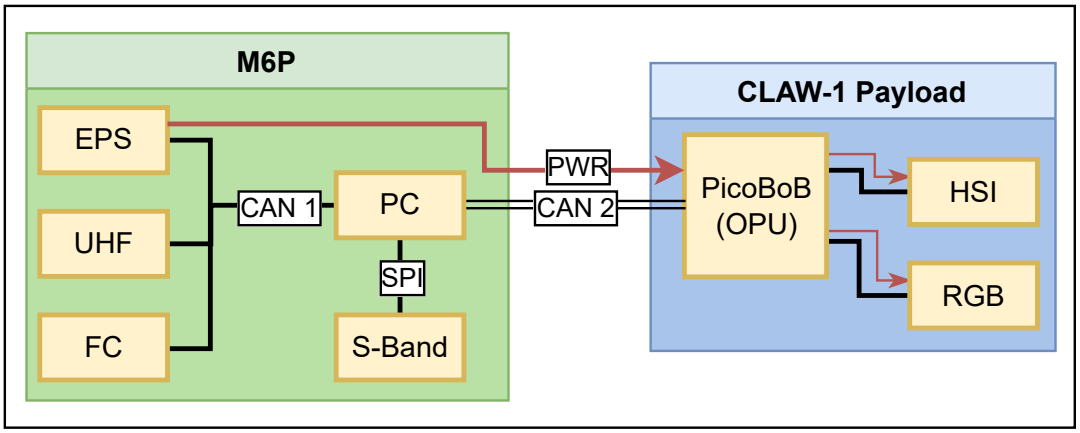


Figure 3: Connections between modules inside the HYPSO-1 satellite

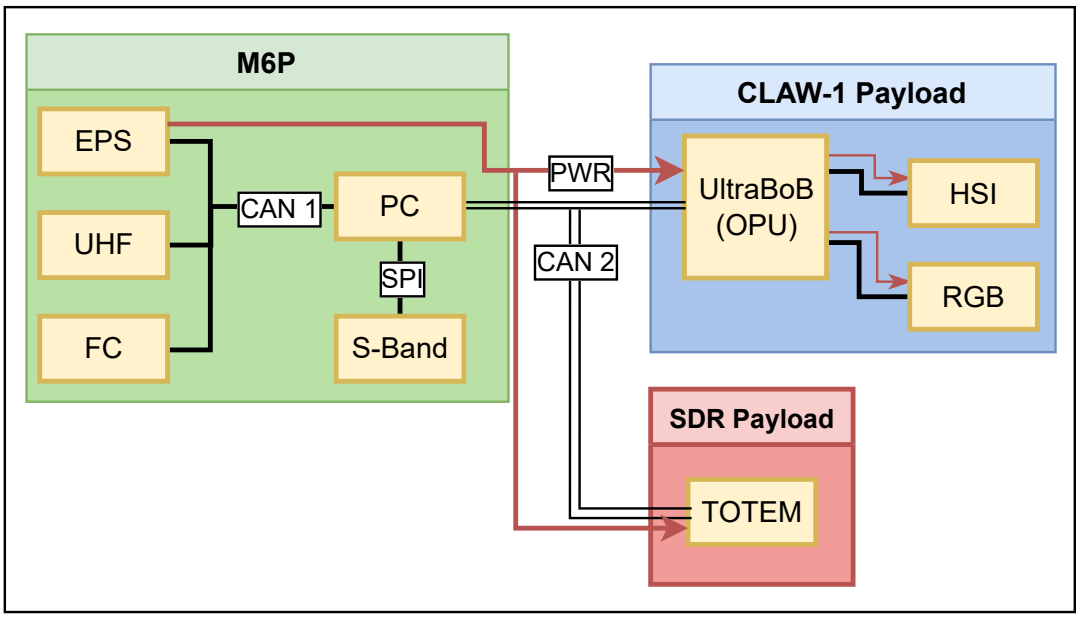


Figure 4: Connections between modules inside the HYPSO-2 satellite

The different modules of the M6P satellite bus are as follows:

- **PC:** The Payload Controller controls the interfaces between the payload and the satellite platform. This includes the CAN bus and the power connections from the EPS.
- **EPS:** The Electrical Power System provides and regulates power to other subsystems. This power is collected from the solar panels, and stored in batteries, until needed. In addition, the EPS also contains a fail-safe mechanism making it avoid electrical damage in both itself and other subsystems.
- **UHF:** The Ultra-High Frequency radio communicates with the Ground Station (GS) through the UHF-band.
- **FC:** The Flight Computer collects data from the sensors and GPS, and uses this to perform activities related to the ADCS.
  - **ADCS:** The Attitude and Determination Control System uses information from sensors to ensure the satellite is in a desired orientation and maintain it. When taking hyperspectral images, the ADCS will ensure the satellite points to the desired location, through the use of actuators.
  - **GPS:** The Global Positioning System is the navigation system of the satellite.
- **S-Band:** The S-band radio communicates with the GS through the s-band. This gives it larger bandwidth and throughput than the UHF, but it requires the satellites to point directly at the Ground Station (GS) when communicating.
- **Solar Panels:** Used for collecting energy from the sun to the satellite

More information on the M6P satellite bus can be found on NanoAvionics's website [2].

The modules of the payloads for the satellites are as follows:

- **OPU:** The On-board Processing Unit contains an on-board processing breakout board, and interface to the satellite payload. The board for HYPSO-1 contains a PicoZed, while HYPSO-2 contains an UltraZed.
- **HSI:** Hyper Spectral Image (HSI) used for taking hyperspectral pictures.
- **RGB Camera:** Used for taking RGB pictures, that helps with georeferencing and registering of the images obtained from the HSI camera.
- **TOTEM:** TOTEM is a high-performance nanosatellite Software Defined Radio (SDR) platform, used for quickly deploying multiple SDR applications [51]. This module is not included in HYPSO-1.

2.2 Breakout Board (BoB) and OPU

As the breakout board for HYPSO-2 is still under development at NTNU, work done in this thesis occurred with a breakout board similar to the HYPSO-1, namely Breakout Board v3 (BoBv3).

The BoBv3 is used for connecting routing signals and power between the M6P satellite bus, and the rest of the payload. Figure 5 displays the interface connections to and from the BoBv3. The HSI camera is connected to the BoBv3 through an 8-pin HIROSE cable for power transfer and flash signal, and an RJ45 Ethernet cable for data and command transfer [44]. The flash signal is used by the HSI to signal the beginning/end of a capture [21]. The RGB camera is connected to BoBv3 through a USB 2.0 mini-B cable used for power, data, and command transfers.

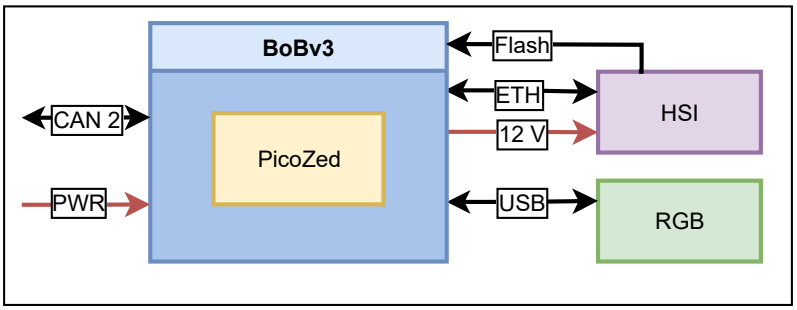


Figure 5: HYPSO-1 payload hardware with interface connections. Note: CAN connections is numbered 2, and corresponds to the same CAN connections as in other figures in this thesis.

The BoBv3 contains a PicoZed, used as the computing unit for the OPU. The PicoZed is a module developed by Avnet, and equipped with a multicore Zynq-7030 System-on-Chip (SoC) from Xilinx, and can be seen in Figure 6.

It combines a Processing System (PS) part with a Programmable Logic (PL) part. The PS contains a dual ARM core and the PL contains multiple Look-Up-Tables, Flip-Flops, adder circuits, block Random Access Memory(RAM) units, and Digital Signal Processing blocks [31]. Complete specifications for the PicoZed can be found in Appendix C.

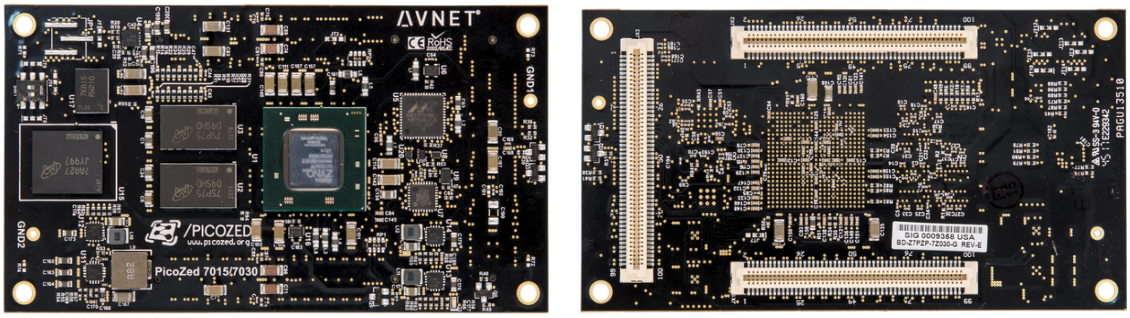


Figure 6: Avnet PicoZed SoC, used for on-board processing on HYPSO-1, front (left) and back (right) angles. The board utilizes a Zynq-7030 SoC from Xilinx [4].

2.3 Network Communication

An understanding of the different communication protocols used in this project can be gained by looking at the Open Systems Interconnection (OSI) model. The OSI model is a generic, protocol-independent model, which is used to describe all form of network communication, and their functions, in a network system [13]. The model splits the flow of data into seven abstraction layers, see Table 1, to describe communication from the physical implementation of transmitting bits across a communications medium to the highest-level representation of data of a distributed application. By implementing layers the systems provides structure to the network designers when they are designing protocols. In the OSI model, each layer provides a service of functionality to the layer above, and is serviced a functionality by the layer below.

#	Layer	Function
7	Application	Used by end-user software, and provides protocols that allow software to send and receive information and present meaningful data to users.
6	Presentation	Defines how to devices should encode, encrypt and compress data. It also takes any data transmitted from the application layer and prepares it for transmission over the session layer.
5	Session	Responsible for opening communication channels, and ensure they remain open and functional while data is being transmitted.
4	Transport	Break data sent in the session layer into segments on the transmitters end, and reassemble it on the receiver end, as well as controlling the rate of flow of data.
3	Network	Break of segments into data packets, and reassemble the packets on the receiving end, as well as routing packets across a physical network (usually done by the use of network addresses).
2	Data Link	Establish and terminate connections between two physically-connected nodes on a network. Data Link also break up packets into frames and transmits them across the physical connection.
1	Physical	Physical cabling or wireless connection between multiple nodes in a network, and is responsible for the transmission of the raw data.

Table 1: Seven-layer Open Systems Interconnection (OSI) network communication Model.

The HYPSO satellites contain a number of protocols, in different layers, that are relevant for this thesis. These are the CSP protocol, CAN protocol, USB protocols, and the Ethernet protocols. Ethernet, and USB, both occupy layer 1 (Physical) and layer 2 (Data Link), and are used for the connection between the HSI and RGB camera respectively. The HSI is connected to the BoBv3 through the RJ45 Ethernet cable, which corresponds to the Physical layer, and uses the Ethernet protocol for transmission of data between them, which corresponds to the Data Link layer. The RGB is also connected to the BoBv3 in the same fashion as the HSI. But the RGB camera uses a USB 2.0 cable in the Physical layer, and the USB protocol for data transmission in the Data Link layer.

2.3.1 CSP

CubeSat Space Protocol (CSP) is a small transport layer protocol specifically designed for easing the communication between distributed embedded systems in smaller networks, like in CubeSats [35]. CSP aims to give some of the same features as TCP/IP, which is used on the Internet, but without having the overhead that comes with the IP header. The CSP allows all subsystems to provide their services on the same network level, without requiring any master node. Each node in the network contain a unique CSP address, which allows every subsystem to communicate with each other. In the HYPSO satellite, the payload and its subsystems all employ CSP as their external communication protocol for communication with the M6P satellite bus. The M6P bus also utilizes CSP internally between submodules, as well as externally when communicating with the Ground Station (GS) [45].

2.3.2 CAN

Controller Area Network (CAN) is an asynchronous serial communications protocol, supporting distributed real-time control (bit rate up to 1Mbs in networks up to 40 m), defined by the ISO 11898:2003 standard [9]. The CAN protocol exists in both the Physical layer and the Data Link layer. In the Physical layer it provides a physical connection between module/nodes using a twisted two pair cable, and the Data Link layer it provides the messaging between nodes with

the transmission of frames. There are four types of CAN messages(frames); Data Frame, Remote Frame, Error Frame, and Overload Frame. The Data Frame is used for sending messages from the transmitter to other modules on the bus. Figure 7 contains a standard Data Frame. The meaning of the bit fields in the figure are as follows:

- **SOF:** Signals the start of a message, and is used to synchronize the modules on a bus after being idle.
- **Identifier field:** Identifier sets the priority of the data frame, while Remote transmission request (RTR) defines the frame type (Data Frame or Remote Frame).
- **Control field:** Identifier extension bit (IDE) sets the Data Frame (Standard or extended), R is a reserved bit, and Data length code (DLC) informs how many bytes in DATA is transmitted.
- **Data:** Contains user defined data to be transmitted (0-8 bytes).
- **CRC field and ACK field:** Protects the message with a checksum using Cyclic redundancy check (CRC) followed by a Delimiter (DEL) bit. Based on the result of the CRC, the receiver acknowledges positively or negatively in the Acknowledge (ACK) slot, which is also followed by a DEL bit.
- **EOF:** Signals the end of the message.

There are two types of Data Frames, CAN 2.0A (standard) and CAN 2.0B (extended). Standard CAN contain an 11-bit identifier in the arbitration field, while extended contain a 29 bit identifier.

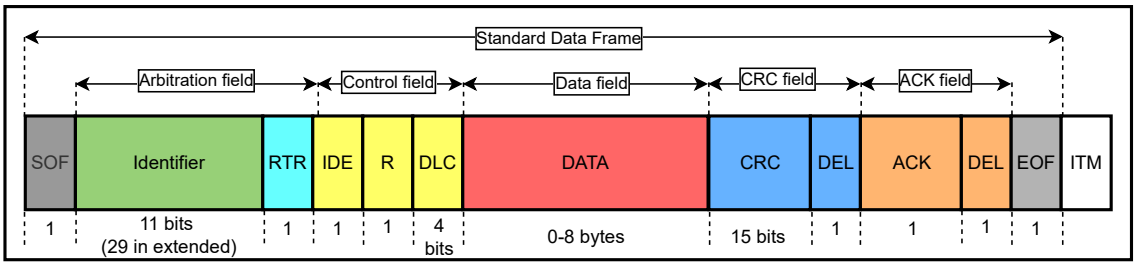


Figure 7: Standard CAN Protocol Data Frame

On the HYPSO satellites there are two CAN-busses, CAN 1 and CAN 2. Figure 3 and Figure 4 shows the CAN connections in the HYPSO-1, and the planned HYPSO-2 satellite respectively. CAN 1 connects the subsystems of the M6P bus together, while CAN 2 connects the payloads to the PC, which acts as a router between the two networks. The CAN network allows communication between the different subsystems in the satellites by sending CSP packets encapsulated in CAN messages.

2.4 HYPSO software

The code developed in this thesis, is part of the code under version control on GitHub for the NTNU SmallSat Lab Organization. GitHub is described in Section 4.2. In this Organization, the author has worked on three different repositories. The first repository, `hypso-sw` [48], is the main repository for the HYPSO team, and contains the executables used on the HYPSO satellite. In this repository the author has worked on setting up GitHub Actions for automation of the test framework. More on GitHub Actions can be found in Section 4.3. The second repository, `assembly-integration-test` [46] repository is where most of the code the author has written resides in, and is where most of the test framework is located. The third repository, `hypso-sw-build-check`[49], is a repository that is used in tandem with GitHub Actions to check building of the software in the hypso-sw repository. For this thesis, a fourth repository is also relevant, namely the `Hardware-in-loop` [47] as it is used in the current test framework. More information on this repository is described in Section 5.

2.4.1 Hypso-sw

The following is based on the authors experience using the repository, in addition with the repository itself [48]. The `hypso-sw` repository, Figure 8, contains source code used for building the executables used for communicating with and running on the satellite. These executables are as follows:

- hypso-cli
- m6p-time-sync
- opu-services
- packet-dropper
- sdr-services

From these services, two are relevant for this thesis, `hypso-cli` and `opu-services`. The `hypso-cli` executable is a Command Line Interface that is used for communicating with the HYPSO satellite using CSP messages. The `hypso-cli` source file will parse the command line inputs and create CSP packets that will be distributed over a CAN network in the satellite. More in depth explanation of `hypso-cli` can be found in Section 2.4.2 `Opu-services` is a monolithic executable running on the OPU, this program has the function of interpreting the CSP packets received from `hypso-cli`, then perform several actions according to what is requested in the packet. These actions will use one or multiple of the services found in Table 2.

Services	**Application**
CSP	Accept and respond to CSP packets.
File transfer	Manage file operations.
RGB	Capture and process RGB images.
HSI	Capture and process HSI images.
Shell	Accept and execute command strings in the OS shell.
TM	Handle request related to telemetry.

Table 2: Services present on OPU

The toolchain used for compiling these executables are encapsulated inside a Docker container, see Section 4.1. The executable can be built for different architectures inside the Docker container with the commands found in Table 3

Command	**Application**
make	Compiles hypso-cli and opu-services for x86 architecture
make ARCH=arm	Compiles hypso-cli and opu-services for ARM architecture
make test	Runs tests from top level
make clean	Removes all compiled source files and other files generated by compilation

Table 3: Commands when compiling hypso-sw

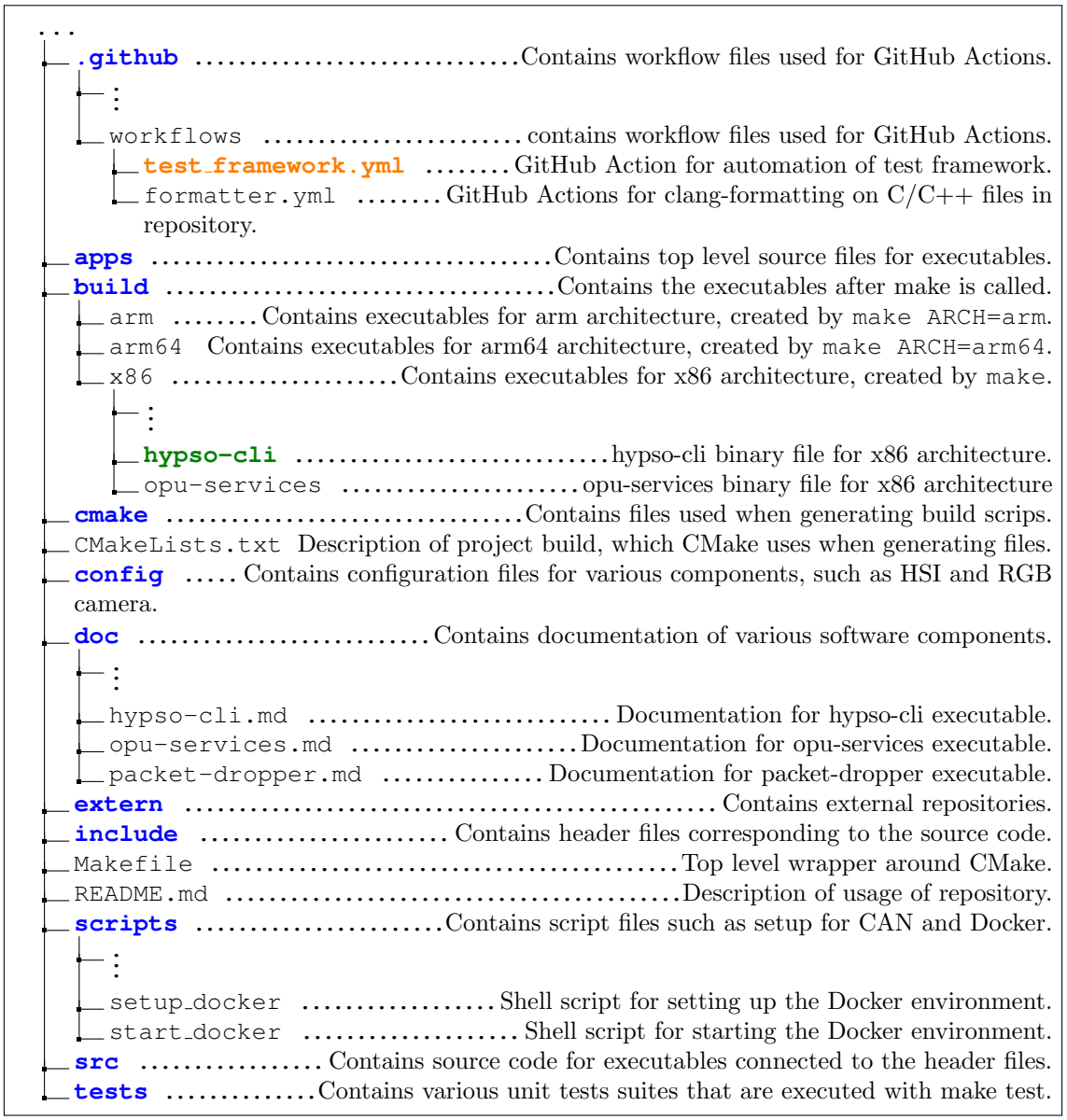


Figure 8: Structure of `hypso-sw` repository. `Test_framework.yml` in orange and `hypso-cli` in green will both be further discussed in this thesis.

2.4.2 Hypso-cli

`Hypso-cli` is a Command Line Interface (CLI) implemented as a Read-eval-print-loop (REPL), which means that it takes readable text commands as input, executes them, and presents the results or output [31]. Figure 9 shows a model of how `hypso-cli` can be used to connect to the OPU in the satellite.

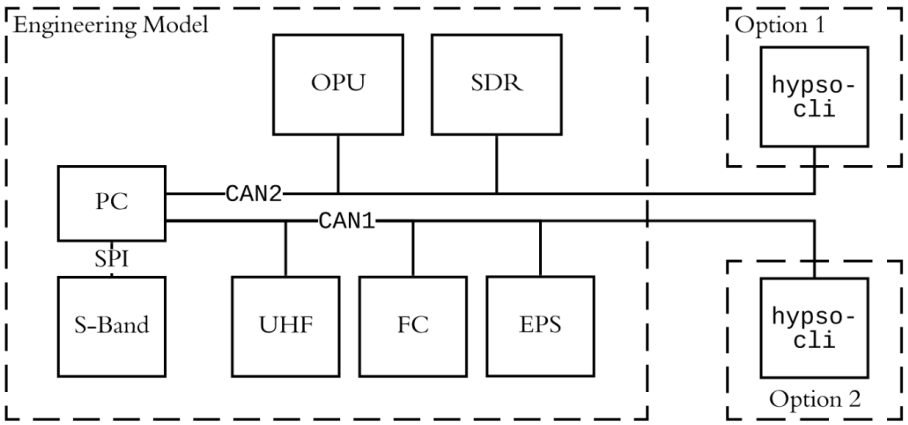


Figure 9: A model of how `hypso-cli` can connect over CAN-bus to communicate with the HYPSO payloads [48]

`Hypso-cli` will parse the command line inputs and evaluate it against a known set commands and sub-commands before creating CSP packets that will be distributed over a CAN network in the satellite.

Table 4 contain a set of generic commands, while Table 5 contains more advanced commands used that are connected to different services.

Generic commands	**Description**
clear	Clear the terminal
help	Print helptext for a command.
list	List commands, or sub-commands of a specific command.
ls	ls -l –color=always
q	Exit this CLI.
shell	Run a local shell command. Enter 'help shell' for subcommands
shell remote	Enter remote shell mode for a specified node.

Table 4: Generic commands in `hypso-cli` application

Command Types	**Description**
csp ...	CSP-specific commands.
eps ...	EPS specific commands.
ft ...	File Transfer specific commands.
hsi ...	CLAW-1 specific commands.
opu ...	Commands for controlling the OPU.
pl ...	Commands for controlling the OPU.
rgb ...	RGB specific commands.
sdr ...	Commands for controlling the SDR.

Table 5: Command types for `hypso-cli` application

All the advanced commands, contains further sub-commands, where the sub-commands are implemented as trees. In Figure 10, sub-commands are displayed as branches going to the right. While

separate commands/sub-commands are displayed as branches going down. A complete set of the commands `hypso-cli` is able to interpret can be found in Appendix A.

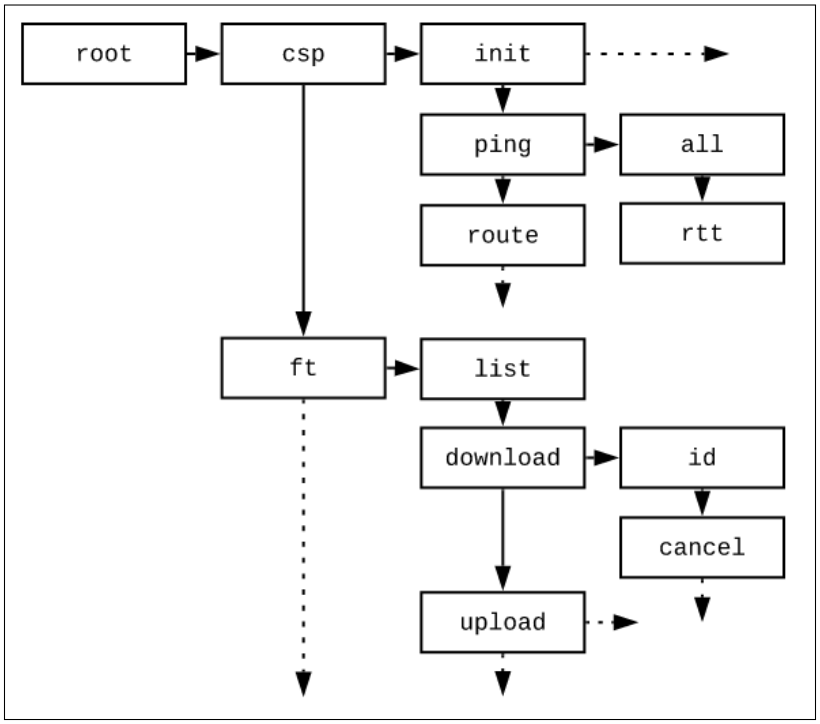


Figure 10: Implementation of hypso-cli sub-commands [48]

2.4.3 Assmebly-integration-test

As with the `hypso-sw` repository in Section 2.4.1, the following information in this Section is based on the authors experience using the repository, in addition with the repository itself [46].

The `assembly-integration-test` repository contain functional and integration tests, see Section 3.2, that interact with, and executes multiple commands on, `hypso-cli` to test the on-board software of the payload.

Figure 11 contain the structure for the repository. `Hypso-cli` is marked in green in this figure. The tests in this repository uses the `hypso-cli` executable, that is created from the `hypso-sw` repository in Section 2.4.1, to communicate with the satellite. For this thesis, the files in the directory `automated_test` are quite relevant as they, in addition with the files `auto_test_execution.py` and `auto_test_sequence.csv` in the `test_menu` directory that the author has created for a previous project, contain the code for the test framework. The files in the `test_menu` directory are used for combining multiple tests together into one larger test, which helps with automating testing, more information on the test menu can be found in Section 2.5.1. The files `test_function.py` and `test_settings.py` in the `tvac_automation` directory contain general functions and setting that is used for interacting with `hypso-cli`.

```
...
├── automated_test
│   ├── template.csv
│   ├── template.log
│   ├── template_result.csv
│   ├── template_settings.json
│   └── test_script.py
├── fps_capability_test
├── hypso-cli
├── image-42.ub
├── image-43.ub
├── pc-buffer
├── README.md
├── test_autmation
├── test_menu
│   ├── ⋮
│   ├── test_menu.py
│   ├── auto_test_execution.py
│   └── auto_test_sequence.csv
├── tvac_autmation
│   ├── ⋮
│   ├── test_functions.py
│   └── test_settings.py
└── ui_less
```

Figure 11: Structure of `assembly-integration-test` repository. `Hypso-cli` in green is the executable created by building the code in `hypso-sw` repository 2.4.1, while `test_script.py` is the main test script for the test framework created in this thesis.

2.4.4 Hypso-sw-build-check

The `hypso-sw-build-check` repository [49] is used in tandem with GitHub Actions, Section 4.3, for checking that the code in the hypso-sw repository is able to correctly build the executables. The file `Dockerfile` is used for setting up the Docker environment, and ends by invoking the `entrypoint.sh` file, which builds the executables in `hypso-sw`, and checks if they succeeded.

```
...
├── Dockerfile
├── LICENSE
├── README.md
├── action.yml
├── check.sh
└── entrypoint.sh
```

Figure 12: Structure of hypso-sw-build-check repository.

2.5 Test Setup

The testing of the test framework created during this project was done on the HYPSO test setup. Figure 13 contains schematic of the test setup. The test setup consists of four main parts; `LidSat`, `Totem`, and `Operational Computer` located at the SmallSat Lab in Trondheim, and `FlatSat` located at NanoAvionics in Vilnius. Together these parts contain most of the modules used in HYPSO-1, and that are going to be used in HYPSO-2 The Operational Computer is a machine

used for executing `hypso-cli`, which communicates with LidSat and Totem through a CAN-bus. Totem is the SDR platform which is going to be implemented in HYPSO-2 [39]. `LidSat` contains the main parts of the payload for the current HYPSO satellite, mounted on the lid of an ESD box [40]. In addition to the payload, `LidSat` also includes the PC and EPS of the M6P satellite bus. The EPS is used for testing and controlling power status of components in the payload, while the PC is used for connecting the payload to the rest of the M6P satellite bus, including a virtual CAN-bridge over the internet to the FlatSat.

The user can connect to the Operational computer through the internet using Secure Shell (SSH). Once connected they can execute `hypso-cli`, which allows control of the rest of the test setup.

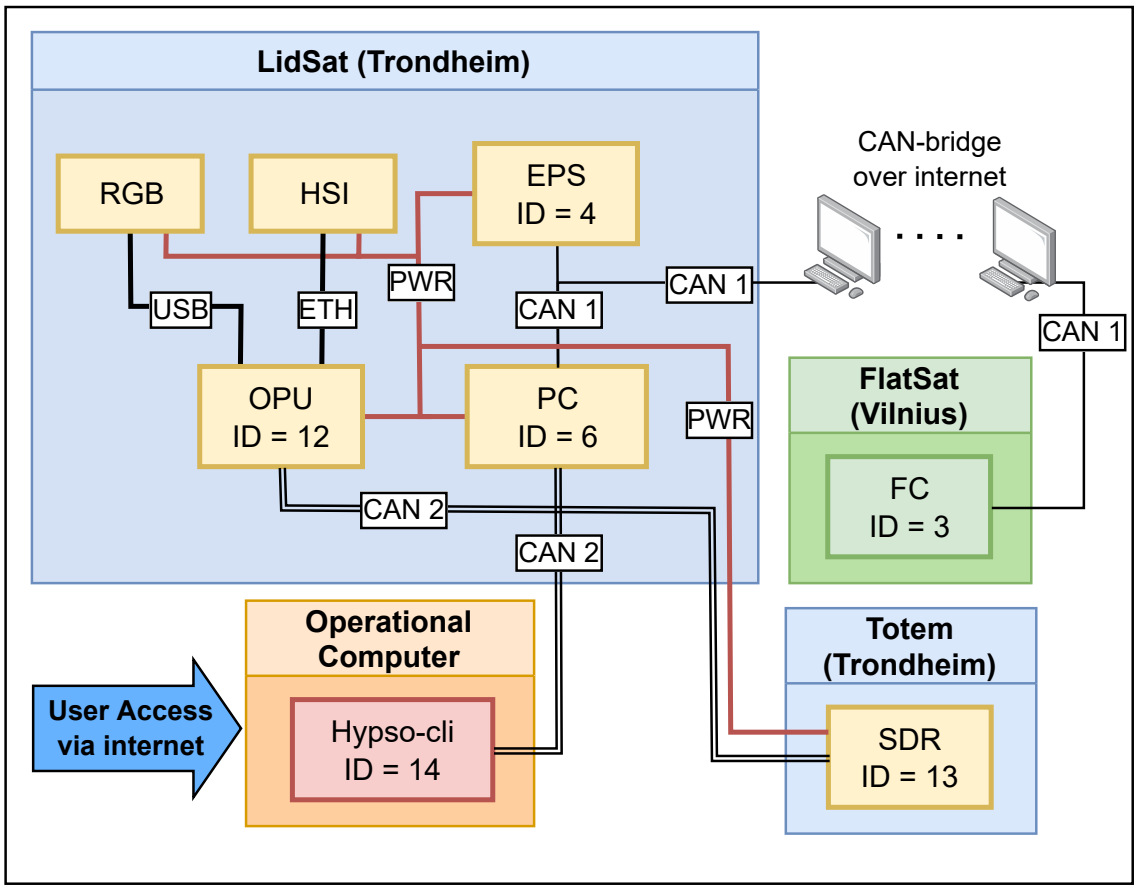


Figure 13: Schematic overview of HYPSO test setup.

2.5.1 Test Menu

The author has previously created a Test Menu system that can be used for both executing and combining multiple test scripts together into one larger test [6]. The python executable for this menu system is located in the `assembly-integration-test` repository under the name `test_menu.py`, and can be seen in Figure 11. The code for `test_menu.py` can be found in Appendix E and Appendix G.

Figure 14 displays the menu, containing three different tests. Each of these tests will be executed, in sequential order, with their arguments. Through the use of the menu, an `auto_test_sequence.csv` file can be created. This file will contain every test with their arguments, and is used when executing the `auto_test_execution.py` python file. The `auto_test_execution.py` file, which can be found in Appendix F, executes every test located in `auto_test_sequence.csv`. Once all the tests have completed, it will give an output where the pass rate of every test together, and tests that failed to pass are displayed. In Figure 15 the output from the tests in Figure 14 is displayed. In this Figure, test2 did not pass, and is therefore displayed in the output.

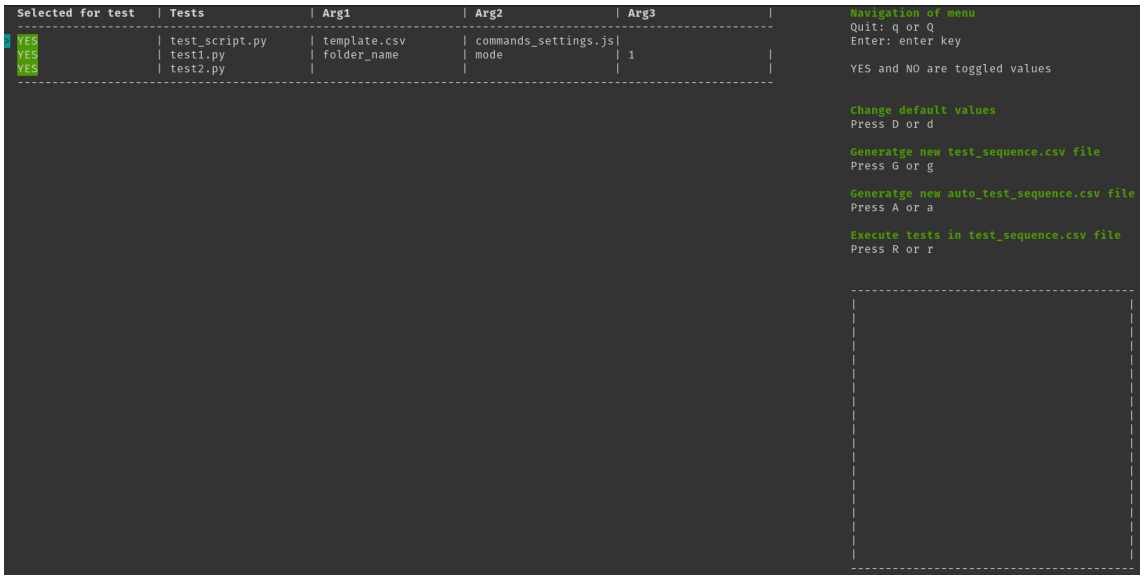


Figure 14: Test menu containing three different tests.

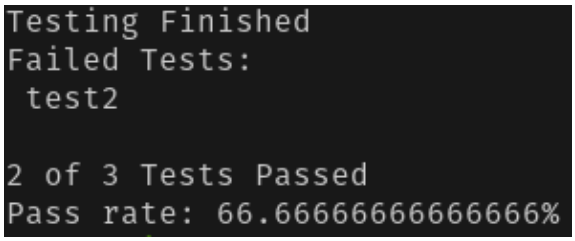


Figure 15: Output from auto_test_execution.py file using tests from Figure 14

3 Test Frameworks

This section contains the background related to testing. The types of testing that exists, and the different kinds automated test frameworks that can be utilized.

3.1 Definition of testing

In the book *Guide to the Software Engineering Body of Knowledge*, P. Bourgue and R.E. Fairley gives a definition of software testing:

*"Software testing consists of the **dynamic** verification that a program provides **expected** behaviors on a **finite** set of test cases, suitably **selected** from the usually infinite execution domain"* [7, pp. 4–1]

In the above definition, bold words correspond to key issues in describing Software Testing Knowledge.

Dynamic testing implies that testing the execution of programs is done with specific inputs predefined by various test cases. As opposed to static testing in which software is tested without code execution, usually done by manual or automated reviews of code, requirement documents and document design [24]. As even a simple program can theoretically contain such a high number of test cases, that complete testing of all cases could take months or years to execute[7]. Testing every possible case is usually not cost-effective due to limited resources and schedules, instead a **Finite** set of test cases are used, which are a subset of all possible test cases. The test cases that have been **selected** are determined by risk and other prioritization criteria to provide high test coverage, with as few as possible amount of cases. Different selection criteria may yield vastly different degree of effectiveness. Identifying the best selection under given conditions tend to be a complex problem [7]. When testing it must be possible to decide if the observed outcome of a test are in line with what is **expected**. If it is not possible to decide if the outcome is acceptable or not, then the testing effort is useless.

3.2 Test types

Software testing can be broken down into two different types: functional testing and non-functional testing. Different aspects of software testing of an application, require different types of testing, such as performance testing, integration testing, unit testing, and so on [36]. Each of these testing types can be used to offer different insight into an application, from code to user experience. There are two main subsets of testing; functional Testing and non-functional testing.

3.2.1 Functional Testing

Functional testing encompasses tests that check critical features, functionality and usability, and are used for ensuring that software features and functionalities are behaving as expected without any errors [26]. It mainly validates the entire application against specifications mentioned in the software requirement documents. Types of testing include unit testing, integration testing, and many more.

While functional testing is a subset of testing, it is also commonly referred to as a specific type of testing, namely black-box testing. In black-box testing one is not concerned with the codebase of the application [25], but rather the input and output and how they conform to the software requirements and specifications. In Figure 16, functional testing is at the top of the pyramid. To optimize return on investment, the codebase should have the least amount of functional tests, as these are usually more time-consuming to develop and execute [29]

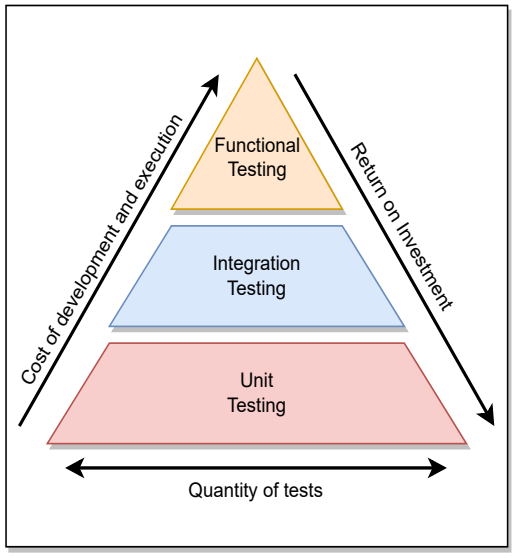


Figure 16: Functional testing types

3.2.2 Unit test

Unit testing are used for verifying individual functionality in isolation from software elements that are separately testable [7]. Depending on the context/software, these can be individual modules or a larger component made of highly cohesive units. Unit testing are typically conducted by the programmer that wrote the tests to ensure the module/larger component is working as intended. Usually to optimize return of investment, the codebase should have as many unit tests as possible, as seen in Figure 16. As they are both quicker to create and execute. As well as making it easier to weed out modules that work as intended, when fixing errors in the codebase [29].

3.2.3 Integration test

Integration testing is the practice of testing the interaction among multiple software components/-modules together. As seen in Figure 16, integration tests take more time and effort to both create and execute, than unit tests, which is why there tends to be fewer of them. But, where unit testing can verify that a module works as intended, integration testing verifies that a module fits in with the rest of the codebase [29]. Integration testing are often ongoing throughout the development of an application. During which software engineers abstract away from the lower-level perspectives, and instead concentrate more on higher level. For larger systems, incremental integration strategies are often preferred to putting all the components together at once [7], as the source of errors can be more effortless to discover.

3.2.4 Non-functional testing

Non-functional testing, evaluates the non-functional aspects of an application [27]. These aspects are performance, usability, reliability, and so forth. Whereas functional testing checks that the application functions as it should, non-functional testing checks the readiness of a system.

3.2.5 Performance testing

Performance testing, also known as efficiency testing, goes under non-functional testing, and checks that an application meets up to its specified performance requirements [7]. When testing the performance it will check how an application, to an extent, handles capacity, quantity and response time [27].

3.2.6 Reliability testing

Reliability testing, checks to which extent an application can continuously perform specified functions without failure. I.e. that an application can perform a failure-free operation for a specified time period in a specific environment [23]

3.3 Test automation frameworks

Test automation framework is a platform that provides an environment where one can execute automated test scripts. This platform consists of a combination of programs, compilers, features, tools, etc. [52]. In other words a test automation framework is a set of components that facilitate executing tests and comprehensive reporting of test results. To this end, implementing a successful test automation framework requires equipment, testing tools, scrips and procedures for testing.

A number of different test automation framework exists, which will be explained further in the following SubSections. These are:

- Linear Testing Framework
- Modular Based Testing Framework
- Library Architecture Testing Framework
- Data-Driven Framework
- Keyword-Driven Framework
- Hybrid Testing Framework

3.3.1 Linear Testing Framework

Linear Testing Framework are commonly used on small applications. Testers manually records each step of a process, as seen in Figure 17, which then creates a script that can be automatically played back to conduct the test [1]. The main advantage of this framework is its speed. As the tester does not have to write custom code, it makes it one of the fastest ways to generate test scripts. In addition, as the test scripts are laid out sequentially, it is easy for any outside party to understand how it works.

The downside of this framework is that scripts developed through this method are not reusable. Data used for the test script are hardcoded in, hence the test script cannot be re-run with multiple sets [5]. Additionally, if the data is altered, the test script also needs to be re-written to accommodate for the change. This can make maintenance a hassle as it can add a lot of rework, which also means that this framework is not suitable for scale as the scope of testing expands.

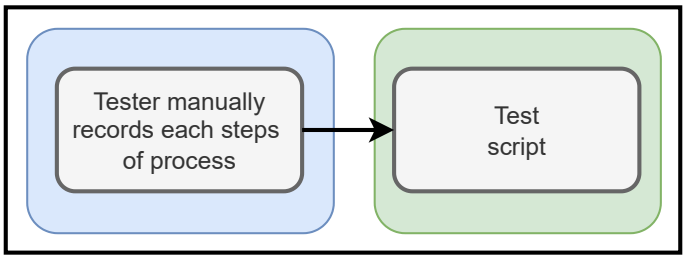


Figure 17: Linear Testing Framework.

3.3.2 Modular Based Testing Framework

Modular Based Testing Framework, has the tester break down the application under test into smaller isolated modules, as seen in Figure 18. These modules can consist of functions, sections, units, etc., each with their own test script [1], which can then be combined into a larger test script in a hierarchical fashion. The different modules in this framework are separated by an abstraction level in such a way that changes made in one module will not yield any affects on the overarching application [28].

By utilizing Modular Based Testing Framework one gains the advantage of low maintenance. If changes are made to the application, only the module and its associative individual test scripts requires alteration. Due to this, this framework also has high scalability [28]. The disadvantage of this framework is that data is still hardcoded in each test, thus whenever one is to test with a different set of test data, it requires changes to be made in the test scripts [1].

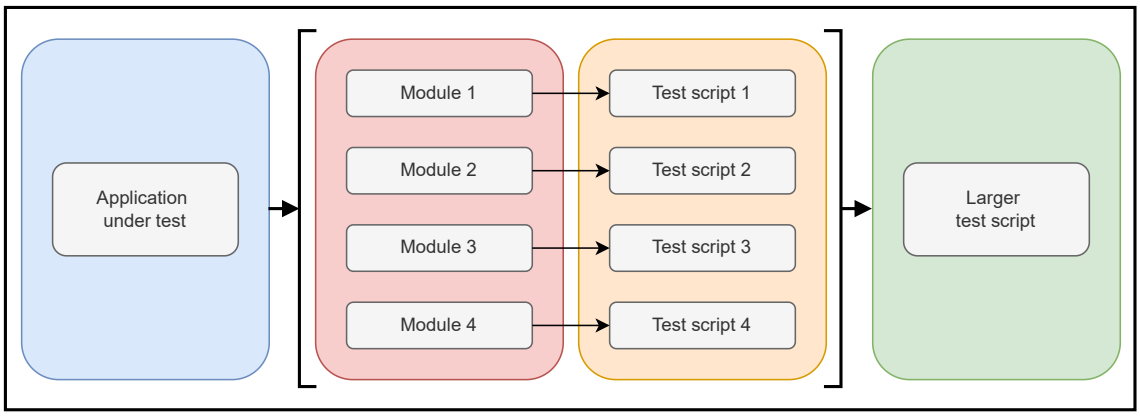


Figure 18: Modular Based Testing Framework.

3.3.3 Library Architecture Testing Framework

Library Architecture Testing Framework is based on the Modular testing framework. But instead of dividing the application under test into modules with various scripts that need to be run, similar task within the scrips are identified and later grouped together by functions [1]. In other words the application is broken down by common objectives/functions. As seen in Figure 19, these functions are kept in a library which can be called upon by the test script when needed. This framework is therefore suitable for applications that contain similar functionalities across different parts of the application [52]. Utilizing this framework leads to a high level of modularization, which in turn makes the maintenance and scalability easier and more cost-effective. In addition, this framework also has a high degree of re-usability, as there is a library of common functions that can be used by multiple test scripts [52].

As with previous frameworks though, the test scripts still contain hardcoded data, and therefore any changes to the data will require changing the test scripts, [28]. The test scripts in this framework

also take more time to develop, and writing and analyzing the common functions within the test scripts requires sufficient expertise on the subject.

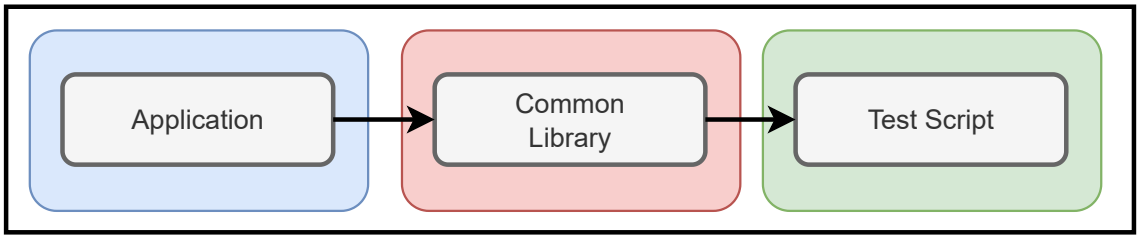


Figure 19: Library Architecture Testing Framework.

3.3.4 Data-Driven Testing Framework

While automating or testing any application, at times it may be required to test the same functionality multiple times with different sets of input data. Therefore, the test scripts shouldn't be embedded with test data, but rather acquire it from an external database outside the test scripts [28]. This is the case in Data-Driven Testing Framework. In this framework the test script logic and the test data are segregated from each other. In Figure 20, the external test data contains both input data and expected data. The input data is used for deciding what the test script will execute, while the expected data is used for verifying the output from the execution was correct.

Using this framework allows tests to execute with multiple data sets, which in turn means that multiple scenarios can be quickly tested with the same test scripts. Setting up this framework can take significant more time than other frameworks, but it saves a lot of maintenance time as scripts won't need to be changed when the test data is changed [1].

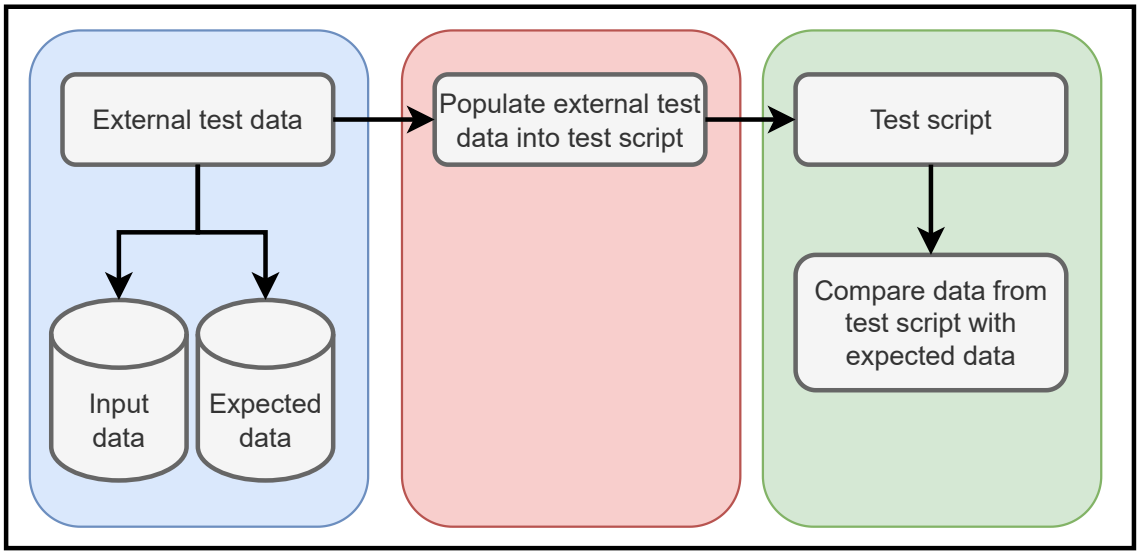


Figure 20: Data-Driven Testing Framework.

3.3.5 Keyword-Driven Testing Framework

Keyword-Driven Testing Framework is an extension to Data-Driven Testing Framework. Instead of only segregating the test data from the scripts, it also keeps certain keywords, representing certain actions, in an external file, as seen in Figure 21 [1]. These keywords are self-guiding as to what actions need to be performed by the application. The keywords and the test data are stored in a table, thus this framework is also known as Table Driven Framework [28].

In table 6 an example of a test case for a generic website using the Keyword-Driven Testing Framework is shown. In this Table keywords like `Login`, `clickLink`, and `verifyLink` have been defined in a step-by-step fashion, where each row represents a new test step. Each of these keywords are associated with certain code in the external keywords file, that will be executed upon use. For each keyword there is also an `Object`. In this example the `Object` contains the location of where keyword-actions is being performed on.

A major advantage to this framework is that a single keyword can be used across multiple test scripts, which makes the code reusable [52]. Additionally, Keyword-Driven Testing Framework doesn't require the user to possess much scripting knowledge, as creating the table is simple. The downside to this framework is that setting up the framework has a high initial cost. As it is both time-consuming and complex. Keywords have to be defined and the object repository/library needs to be set up, [1]. Keywords can also become hard to maintain when scaling the test operation, and new keywords are introduced.

Step Number	Description	Keyword	Object
1	Login to application	Login	Login Button
2	Click on homepage	clickLink	//[@id='homepage']
3	Verify logged-in user	verifyLink	//[@id='link']

Table 6: Example of a test case of a generic website using the Keyword-Driven Testing Framework. This Table is based upon a Table located at [28]

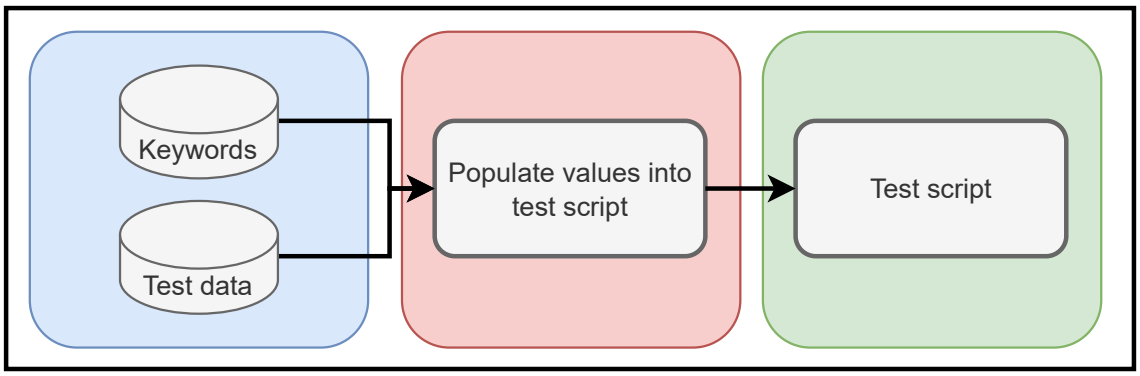


Figure 21: Keyword-Driven Testing Framework.

3.3.6 Hybrid Testing Framework

Hybrid Testing Framework, is as the name suggests, a hybrid combination of multiple testing frameworks, [28]. By combining multiple frameworks, they can be set up to take advantage of each other's positive sides, while mitigating their weaknesses [1].

As every application is different, the process used for testing should also be different. So that it fits within the purpose of the application [5]. A hybrid framework can be more easily adapted to get the best test result for each application while sacrificing as little as possible.

Figure 22 contains an example of a Hybrid Testing Framework. This Framework consists of a Modular Based Testing Framework, combined with a Data-Driven Testing Framework. By combining these two one alleviates the hardcoded data disadvantage from the Modular Based Testing Framework, while keeping the modularity that Data-Driven Testing Framework lacks. One disadvantage with the Hybrid Testing Framework, is that as it combines multiple frameworks it becomes more complex than either of them. Which in turn results in higher initial cost of when creating the framework.

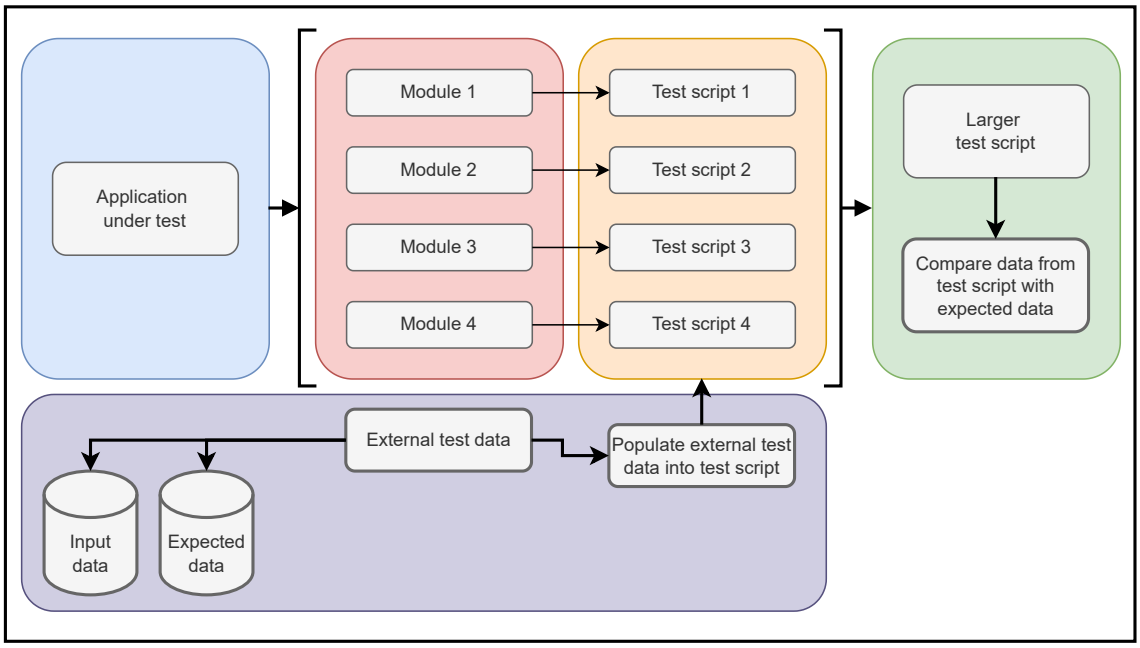


Figure 22: Hybrid testing framework as a combination of Modular Based Testing Framework and Data-Driven Testing Framework.

4 Methods and Tools

This Section contains the tools and methods that have been utilized during the work of this thesis.

4.1 Docker

The HYPSO team utilizes a software platform known as Docker when building the executables in the `hypso-sw` repository. Docker provides the ability to package and run an application in a loosely isolated environment called container [11]. Using Docker, deploying software to different environments become much easier, as one does not have to manually provision necessary software to each machine. Neither does one have to manually configure how each machine utilizes its resources for the software [32].

In the same way a container on-board a cargo ship can contain multiple items that are isolated from the other containers on-board, so can a container in Docker contain various software that is isolated from the rest of the machine. The containers in Docker are lightweight and contain everything needed to run the specific application [11]. This ensures that everyone that has the same container, have the same code and dependencies so that the application runs quickly and reliably from one computing environment to another [12].

For the HYPSO team, using Docker is advantageous as it guarantees that the toolchain for building the software is the same no matter where it is run, or what system it is run on. Setting up the Docker environment is easily done by executing the shell script `setup_docker` located in the `scripts` directory in the `hypso-sw` repository, Figure 8. After the setup has been completed, the Docker container can be opened by executing the `start_docker` shell script located in the same directory.

4.2 Git & GitHub workflow

The software for HYPSO is written by many people, working on a wide number of tasks. To ensure the software continues to work throughout the development and completion of these tasks, the HYPSO team utilizes Git. Git is a free and open-source distributed version control system, that has been designed to handle large projects with speed and efficiency [14]. By using Git, one is able to record changes made to the codebase over time in a database called repository. With these records, one can look back at the history of the codebase, to see what changes were made, when were they made, and by whom were they made. This gives higher security to the codebase, as one is also able to revert the codebase to an earlier version of the if something were to go wrong.

As Git as a service only exists locally on a machine, the HYPSO team utilizes GitHub. GitHub is a cloud-based Git repository hosting service [10]. In other words, GitHub is an online database that allows keeping track of, and sharing of Git version control projects outside a local computer. GitHub expands upon the features and advantages that Git provides. In addition to the version control system from Git, GitHub also provides built-in control and task-management tools. As well as providing a GitHub Marketplace service that can be used to include even more features.

The workflow on GitHub centralizes around a working master branch, that should always be deployable, represented as the thick blue line in Figure 23. To ensure this, GitHub can be set up to prevent changes made to this master branch, until they have been reviewed, tested, and ensured working. This is the setup that the HYPSO team has. Every time someone makes new changes to the master branch, a different member in the team has to review them and sign off that the new changes are working as intended.

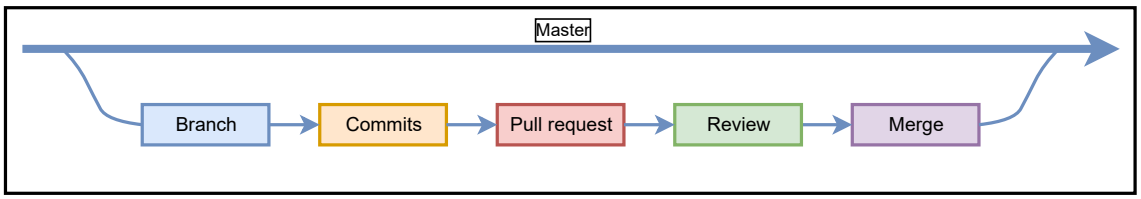


Figure 23: Git & GitHub workflow

4.2.1 Issues

Issues are a task-management feature that is rigorously used by the HYPSO team. Issues are a way for the HYPSO team to inform other people when a feature in the software is missing, or a bug is discovered. Every member in the HYPSO team can create issues. Inside these issues, a detailed explanation is given on why it is created, and then people in the team can use this issue as a place for discussing a possible solution.

4.2.2 Branch

Git and GitHub utilizes a feature called Branches. Branches are a way of copying the codebase into a new, contained, area where one can develop features, fix bugs, and safely experiment with new ideas without altering the main version of the codebase [15]. The master branch (main if the repository is created after October 1, 2020) is the default branch when a repository is created. Following good code practice, this branch should always be deployable, which is why developing code in this branch is not recommended.

The HYPSO team uses branches for development. Every branch is independent of one another, so that changes made in one branch will not affect other branches. This enables the HYPSO team to work on independent tasks for the same systems without interfering with each other. In figure 23 a branch has been created, blue box. This branch can then be used to develop a new feature for the codebase. When the developing has finished, and has been reviewed, the branch can be merged back into the master branch.

4.2.3 Commit

The main advantages with using Git and GitHub is the possibility to track the progress that one has made. To accomplish this, commits are used. A commit is a "save point" within Git's version control system. Every commit made is considered a separate unit of change [22]. In Figure 23 multiple commits have been made inside a separate branch from the master branch, orange box. Each commit work like a separate "save point" that one is able to revert to at will. Every commit made also requires a commit message. A commit message is useful for explaining what has changed in this commit vs. the previous one, and it enables other people to follow and understand the changes that have been made, and why they were made.

When a commit is created, it is saved locally. To update the remote branch created through GitHub, a push has to be made. Pushing is the act of updating the branch on GitHub with all the recent commits made since the last push.

4.2.4 Pull request

When the work done in a branch has concluded, one might want to update the master branch with the new code. To accomplish this one can open a pull request, represented in Figure 23 with the red box. A pull request is a way to inform others about changes that have been pushed to a branch, and get their feedback. When the pull request is created, it starts a discussion about the

commits, and changes, that have been made to the codebase. Here everyone can see what will be changed in the new master branch. A pull request can also be created at any point in time, if one wants to share some general ideas, is stuck and needs help, or when a review of the changes made are needed.

4.2.5 Review

After a pull request has been made, a discussion and/or review session is started, represented with the green box in Figure 23. In this session the person or team reviewing the changes may ask questions or come with some comments about the changes. Pull requests are designed to encourage and capture these conversations in a way that allows everyone on the team to be on the same page when the master branch is updated with the new changes [22].

4.2.6 Merge

When the changes made on a branch has gone through a pull request and review process, and has been accepted by at least one other team member, it is ready to be merged into the master branch. In Figure 23 one can see that this is the final step of the branch, and signifies the end of the workflow. If one were to branch out of the master branch after this merge, then the new branch would include the changes made in the previous branch.

4.3 GitHub Actions

In 2018 GitHub launched GitHub Actions, a continuous integration and continuous delivery (CI/CD) platform aimed at automating building, testing and deployment of pipelines [20]. By utilizing GitHub Actions, one can bring automation directly into the software development lifecycle via event-driven triggers. These event-driven triggers, are specified GitHub events that can range from creating pull request to creating a new branch.

GitHub Actions automation are handled by workflows that defined the automation process. These workflow files are YAML files placed in the `.github/workflows` directory in a repository. In Figure 8 in Section 2.4.1, one can see two workflow files, `test_framework.yml` and `formatter.yml`.

Every workflow file consists of several core aspects:

- **Events:** Event is a specific activity in a repository that triggers a workflow run. This activity can for example be when someone creates a pull request, opens an issue, pushes a commit to a repository and so on. A workflow run can also be triggered on a schedule.
- **Jobs:** Jobs is a set of steps in a workflow file that execute on the same runner. Each job runs in its own virtual machine and parallel to other jobs, unless otherwise specified.
- **Steps:** Steps are individual tasks inside a job. These tasks can either be a script, an action or a command, that will be run. The steps will be run in sequential order, and as every step inside the same job will be executed on the same runner, it is also possible to share data between them.
- **Actions:** An action is a custom application for the GitHub Actions platform. Actions are used to help reduce the amount of repetitive code that is written in a workflow file. For example and action can be used for pulling a GitHub repository, setting up a toolchain for the build environment and so on. GitHub also have a GitHub Marketplace where one can find multiple actions that other people have created.

- **Runners:** A runner is a GitHub Action application that runs workflows when they are triggered. The runner listens for available jobs, runs them, and reports back the progress, logs and results. Each runner can be either hosted by GitHub or they can be self-hosted on a localized server. Self-hosted runners can operate on any operating system, while GitHub hosted runners operate on either Ubuntu Linux, Windows or macOS. Additionally, GitHub hosted runners can not access target hardware, while self-hosted runners can.

Code Listing 1 contains the `formatter.yml` file located in `hypso-sw` repository under `.github/workflows` directory, Figure 8. This file was not created by the author, but it will be used as an example for a GitHub Actions workflow file. In this file one can see that it will be triggered on every push. The file contains one job with the name of clang-format Code Formatter. This job will run on GitHub's own Ubuntu server with the latest version, and contains two steps. The first step, `checkout on branch`, uses the GitHub Action Checkout [18]. This action is used for allowing the workflow access to the relevant repository. Following this the second step, `Clang Code Formatter`, uses a GitHub Action that has been created by the HYPSO team, that executed Clang formatting on every C/C++ file in the codebase. Inside this step there is also an environment variable, env. This environment variable contains a GitHub token, that gives access to the NTNU-SmallSat-Lab GitHub organization, and the repositories that are contained within.

```yaml
name: clang-format Code Formatter
on: push
jobs:
  lint:
    name: clang-format Code Formatter
    runs-on: ubuntu-latest
    steps:
    - name: checkout on branch
      uses: actions/checkout@v3
      with:
        ref: ${{ github.head_ref }}
    - name: Clang Code Formatter
      uses: NTNU-SmallSat-Lab/clang-format-action@master
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
```

Code Listing 1: formatter.yml file located in hypso-sw repository under .github/workflows directory, Figure 8

When a GitHub Actions workflow run is executed, a summary of the results is displayed under the Actions tab on GitHub page, Figure 24. On this page one can see every workflow run, when it was run, and how the result ended up. One can then further go into every specific run, to see the jobs that were run, Figure 25, and inside each job, one can see the specific steps that have been taken, Figure 26. Additionally, when creating a pull request, if there are any workflows set to run, the result from the run will be displayed in the pull request. Figure 27 shows how it will look when a workflow file fails, while Figure 28 shows how it looks when the workflow files passes.

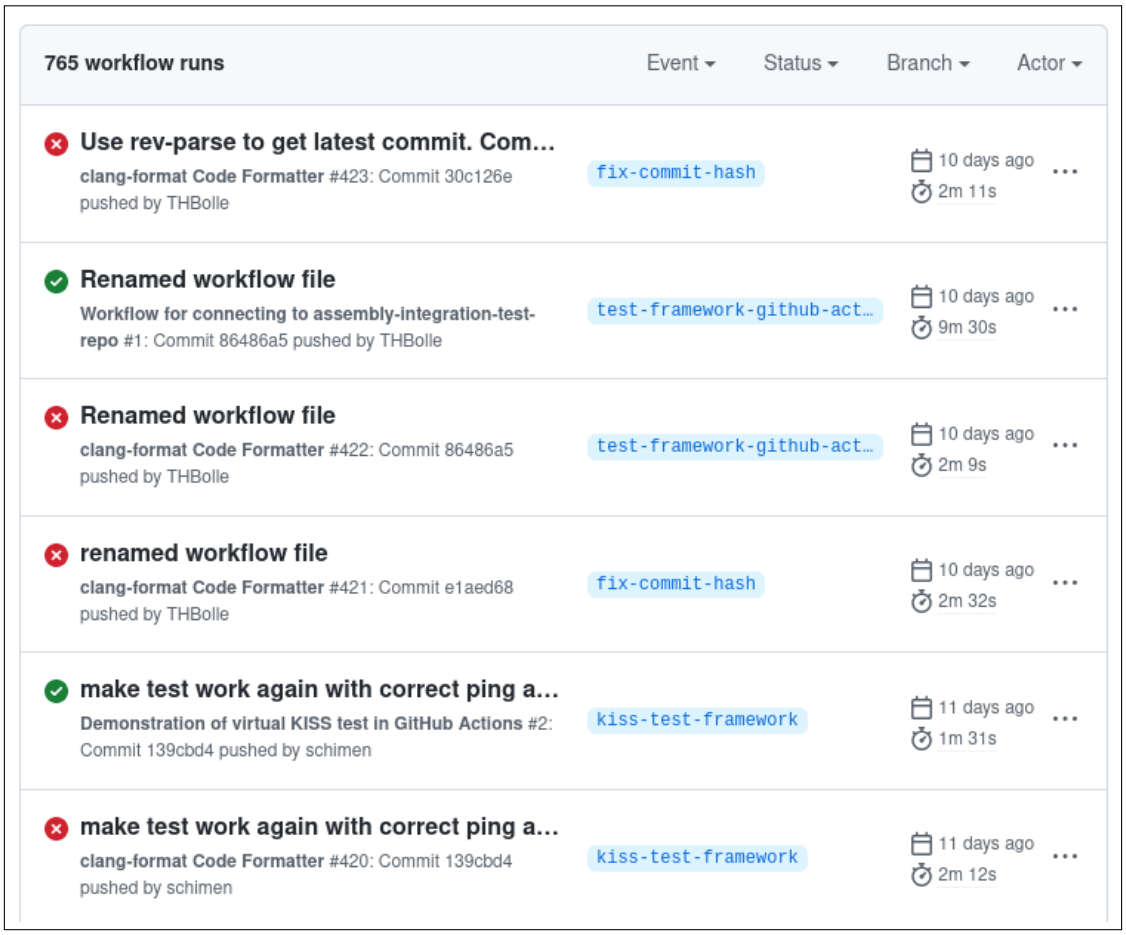


Figure 24: On the GitHub page, one can see the result from previous workflows runs.

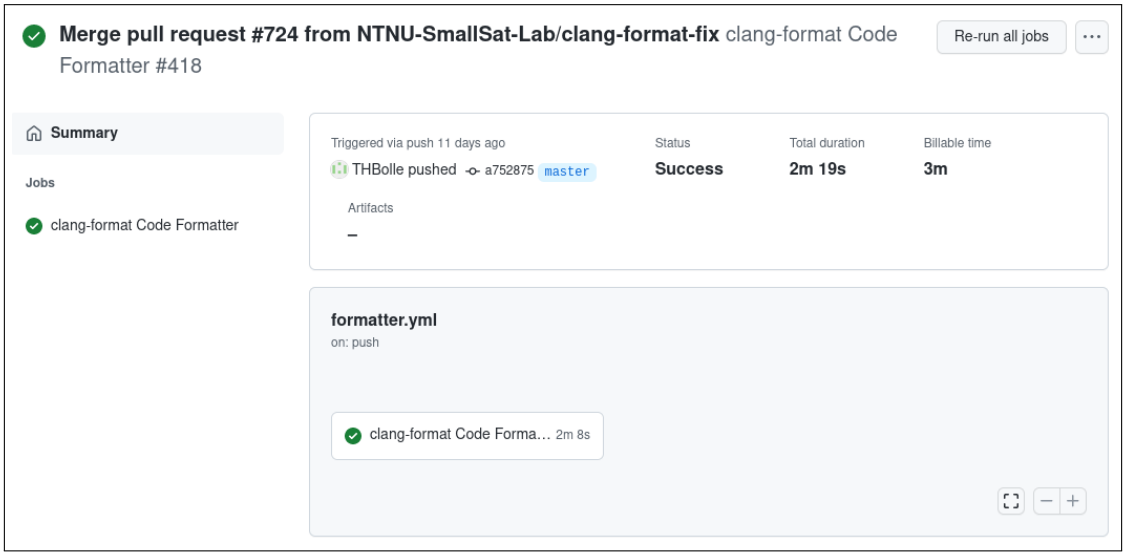


Figure 25: Inside each GitHub Actions workflow run, one can see every job that has been taken for that specific run.

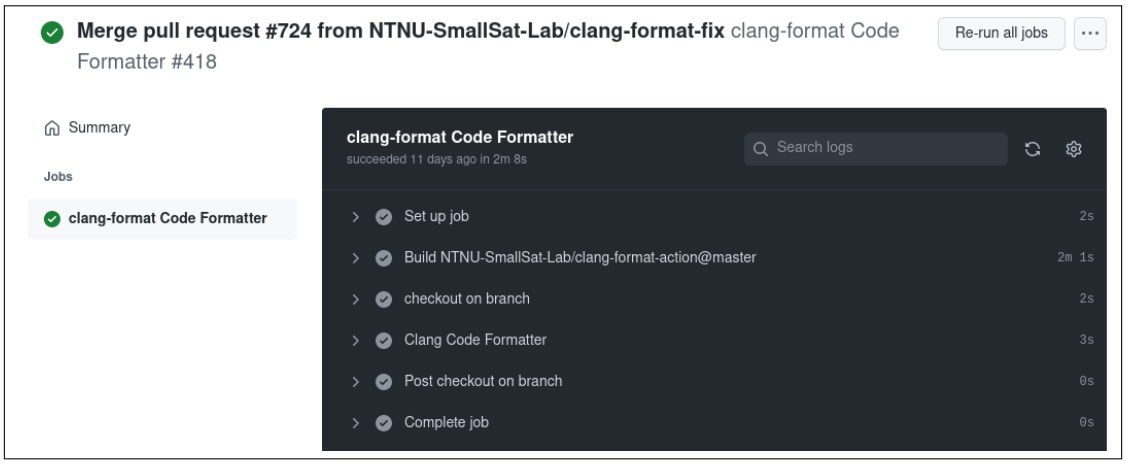


Figure 26: Inside each GitHub Actions job, one can see every step that has been taken for that specific job.

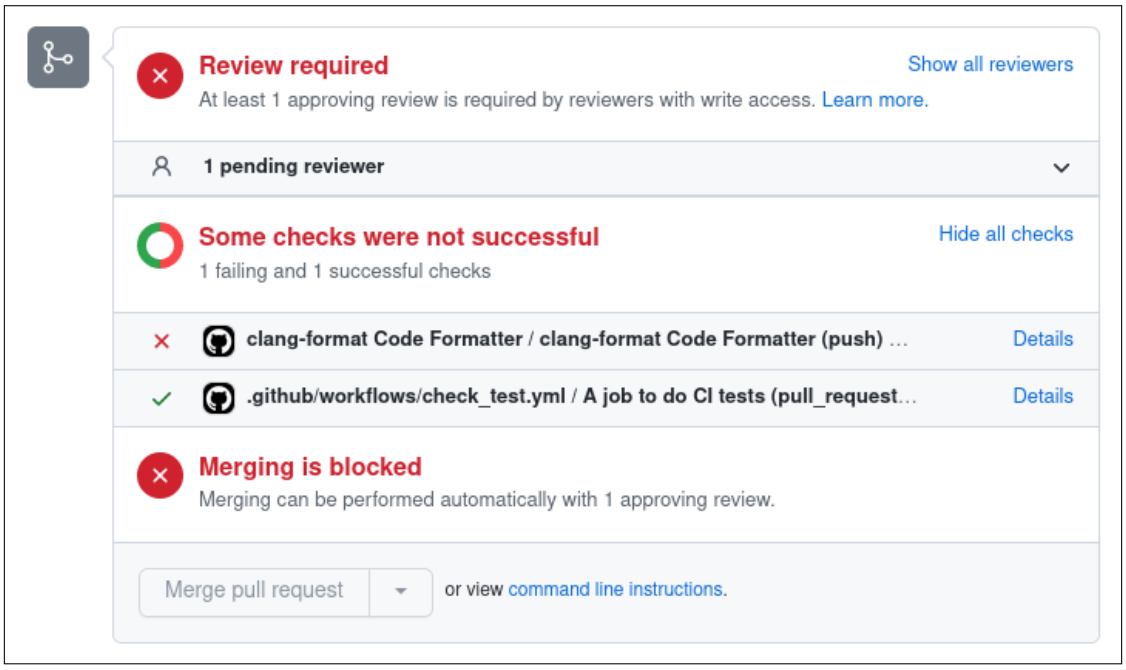


Figure 27: Message in pull request when a triggered workflow file has failed. In this case there were two workflow files that were triggered, where one passed and one failed.

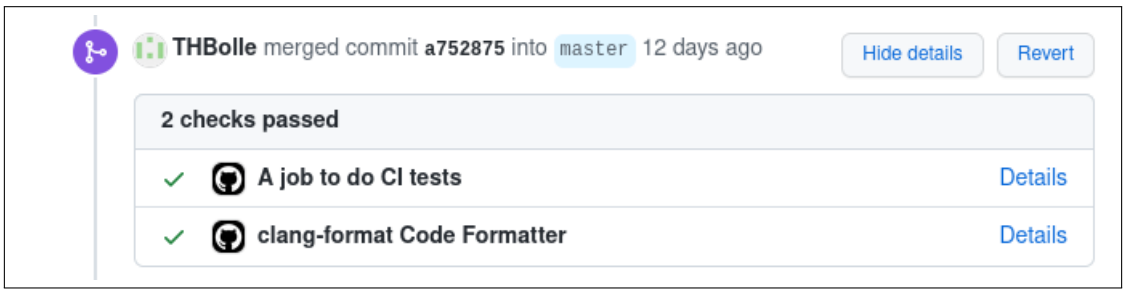


Figure 28: Message in pull request when a triggered workflow file has passed. In this case there were two workflow files that were triggered, where both files passed.

5 Analysis & Requirements

Before the author started working on creating a test framework, an analysis of the current test setup had to be done, to figure out what requirements the new test framework required.

5.1 HYPSO current test framework

The HYPSO team has previously created multiple tests, that execute automatically using Jenkins.

Jenkins is a self-contained, open source continuous integration and continuous delivery (CI/CD) platform, that can be used for automating building, delivery and deployment of software [34]. Jenkins and GitHub Actions, Section 4.3, are quite similar and both can be used for automatic testing. Some main similarities between them are:

- Jenkins uses `Declarative and Scripted Pipelines` when creating workflows, which are similar to the workflow files GitHub Actions utilizes.
- Jenkins uses `stages` to run a collection of steps, while GitHub Actions uses jobs as a way to group one or more steps or individual commands.
- Jenkins and GitHub Action both support container-based builds, like Docker, Section 4.1.
- In both Jenkins and GitHub Actions, steps or tasks can be reused and shared with the community.

Some differences between Jenkins and GitHub Actions are:

- Jenkins uses two different types of syntaxes when creating pipelines, `Declarative Pipelines` and `Scripted Pipelines`, where Declarative is a simplified version of Scripted as declarative syntax is more limited (One is not able to inject code anywhere in pipeline in Declarative). GitHub Actions, on the other hand, uses only one type of syntax when creating workflows and configuration files, YAML.
- Jenkins' deployments are usually self-hosted, while GitHub Actions offers both self-hosted runners and cloud based runners.
- Jenkins is based on accounts and triggers and centers on builds which does not conform to GitHub events, while GitHub Actions can trigger on every GitHub event.

The main difference between GitHub Actions, and Jenkins for the HYPSO team, is how both of them integrate with GitHub. For the HYPSO team, Jenkins is currently set up as a "multi-branch pipeline" in the `hardware_in_loop` GitHub repository [47]. This means instead of branches for development of specific features, each branch is a separate testing pipeline, that each perform a different set of procedures. Thus unlike the other repositories used by the HYPSO team, the hardware_in_loop repository uses neither the Git nor GitHub branching style explained in Section 4.2. Additionally, these branches are not supposed to ever be merged with the master branch.

For the HYPSO team, using Jenkins proved to be an unsatisfactory solution. Since it is not ingrained into the GitHub ecosystem, it meant that if one wanted to create new tests, one had to become familiar with how Jenkins works. Which took a lot of time that many members of HYPSO didn't have due to other school work. This resulted in very few people familiarizing themselves with Jenkins, which in turn resulted in, when Jenkins is not working properly (as of the moment this thesis is written) very few people are able to fix it.

With both Jenkins and GitHub Actions, the tests being automated have to be made beforehand. Currently, the predefined tests that execute on Jenkins have been created manually, by hand, and the tests in the `Assembly-integration-test` repository, Section 2.4.3 have also been created manually, and they are not compatible with multiple sets of data. Creating these tests is the

biggest problem with the current setup. There are no clear recipes on how these tests are to be made, which have resulted in some tests having to be manually executed, while others can be executed through a script. Additionally, creating tests can take a lot of time, which as mentioned before, many team members of HYPSO might not have.

A test framework that eases and streamlines the process of creating new tests, and that can be automated with the use of GitHub Actions would be very advantageous.

5.2 Requirements

From the analysis of the current HYPSO test setup, the author made some requirements that the test framework has to fulfill. In table 7 a summary of the requirements are shown, and following the table the reason for each requirement is detailed.

Req. Number	Description
REQ-0	The test system shall imitate normal usage of hypso-cli.
REQ-1	Creating new tests in the test system shall be quick and effortless.
REQ-2	The test system shall log the results from the tests in a results file.
REQ-3	The test system shall be usable with future features on hypso-cli.
REQ-4	The test system shall have low maintenance.
REQ-5	The test system shall be able to test multiple sets of data with the same command.
REQ-6	The test system shall be able to automatically execute tests through the use of GitHub Actions.

Table 7: Requirements for test framework

REQ-0: The test system shall imitate normal usage of `hypso-cli`. This means that the interaction between the test framework and hypso-cli shall be indistinguishable to a normal user manually inserting commands. The test framework shall be used for ensuring commands on `hypso-cli` function as they should under normal operation, if it didn't imitate normal usage it would have no reason to exist.

REQ-1: Creating new tests shall be quick and effortless. This means that creating new tests shall not require large amount of scripting, and should not take longer time than creating tests without the use of the framework.

REQ-2: The test system shall log the results from the tests in a results file. After the test framework has been used, it shall save the results from the testing in a file. By having test results saved in a file, one can use it to see progress from testing over time.

REQ-3: The test system shall be usable with future features on `hypso-cli`. As more development occurs on the HYPSO satellites more features on `hypso-cli` might be needed. When this occurs one should not have to create a new test framework, but rather the test framework in use should be able to handle new features.

REQ-4: The test system shall have low maintenance. The test framework shall require as little maintenance as possible. As the HYPSO team changes throughout the years, it is not given that there will always be people with ample knowledge of the inner workings of the test framework to be able to change/fix it very easily.

REQ-5: The test system shall be able to test multiple sets of data with the same command. Having to write tests that use the same commands as previous tests, but with new data wastes a lot of time. Therefore, to save time, the test framework shall be able to test multiple data sets of data with the same command.

REQ-6: The test system shall be able to automatically execute tests through the use of GitHub Actions at specified GitHub events. This is to ensure that when the codebase in the master branch is updated with new code, both the new and old code still functions as intended.

5.3 Framework analysis

After the requirements had been set, the author started looking at what framework from the ones explained in Section 3.3 would fit best for HYPSO. All the frameworks end up with a test script that can be executed by itself, and a log file can also be extracted from a test script, which means that no matter what framework is chosen, they can all conform to REQ-2 and REQ-6 in Table 7.

5.3.1 Linear Testing Framework

In the Linear Testing Framework the tester has to manually record each step of the process before being able to automatically run it as a test script. In this framework new tests are quick to create which is very advantageous considering requirement REQ-1 found in Table 7. But as explained in Background section 3.3.1, this framework can require a lot of maintenance which would directly contradict requirement REQ-4. This framework can also not be used for creating tests that utilizes multiple sets of data, which interferes with requirement REQ-5. Based on these grounds the Linear Testing Framework was not selected.

5.3.2 Modular Based Testing Framework

The Modular Based Testing Framework is suitable for when applications under test are divided into modules, see Section 3.3.2. As then each module can have its own test script. This framework provides low maintenance, as a change in one module does not affect other modules. Which is very advantageous considering requirement REQ-4 in Table 7. The Modular Based Testing Framework also has high scalability which works well with requirement REQ-3, as more modules can be added without interfering with existing modules. The downside with this framework, data is still hardcoded in each test, is in direct violation of requirement REQ-5. REQ-5 says that the framework shall be able to test multiple sets of data with the same command. This is not possible when the data is hardcoded in the test scripts. An additional downside, as mentioned, is that this framework works well when the application is divided into modules, which the application hypso-cli, to a degree, is not, see Section 2.4.2. `Hypso-cli` contains multiple commands, that are all executed through the same terminal. Therefore, it makes no sense to split up the commands into their own modules based on their functionalities, and as a result the Modular Based Testing Framework was not selected.

5.3.3 Library Architecture Testing Framework

Library Architecture Testing Framework is a framework that is suitable for applications that contain similar functionalities across different parts of the application, see Section 3.3.3. From this statement it is evident that this framework won't be suitable as a test framework for `hypso-cli`, considering that `hypso-cli` contains a large amount of dissimilar functionalities, see Section 2.4.2. Additionally, this framework also relies on having hardcoded data in the test scripts, which goes against requirement REQ-5, Table 7.

This framework does contain a high level of modularization, which makes the maintenance and scalability easier and more cost-effective, which is good for requirement REQ-4. Nevertheless, due to the drawbacks of this framework, it was not selected as a test framework for HYPSO.

5.3.4 Data-Driven Testing Framework

Data-Driven Testing Framework are useful for testing an application that requires testing the same functionality multiple times with different sets of input data, see Section 3.3.4. This relates well with requirement REQ-5 in Table 7. Due to this framework containing data in data sets separate from the test script, it has low maintenance. This framework contains data in data sets separate from the test script, which gives lower maintenance, which is positive considering REQ-4. Changes made in the test data does not require changes to be made in test scripts. The maintenance in this framework could still be lower though, as adding new features or altering existing features requires the creation of new test scripts. This framework does also not address requirement REQ-1 which states that creating new tests should be quick and effortless, as every command that is to be tested has to be manually entered in the test script. Since the Data-Driven Testing Framework does not ease the creation of new tests, this framework was not selected.

5.3.5 Keyword-Driven Testing Framework

Keyword-Driven Testing Framework improves upon Data-Driven testing Framework by including a set of keywords that correspond to a specific action that the test script will preform, see Section 3.3.5. In the case of `hypso-cli` these keywords would correspond to specific command(s) that would be executed. This framework is well suited for the requirements REQ-1, REQ-4 and REQ-5 in Table 7. When creating new tests in this framework, users have to utilize two files, a file containing a set of test data, like in the Data-Driven Testing Framework, see Section 3.3.4, and a keywords file containing specific code that corresponds to the action the keyword represents. The code in the Keywords file can be set up to automatically use the sets of data found in the data file, which then allows it to be tested over multiple sets of data. With these two files, the user can create test cases very quickly as they only have to create a table like Table 6 in Section 3.3.5. Additionally, users with less scripting knowledge will also have an easier time creating new tests. Which can be very beneficial for HYPSO team, as new members might require less time to familiarize with the test framework before starting to make tests. By having the test cases be tables, when a test has to be altered, the user only has to alter the tables that are used, giving lower maintenance time.

So far the Keyword-Driven Testing Framework is the framework that seems most suited as the framework for `hypso-cli`, but also this framework contains some downsides. The first downside is the initial cost of creating the framework, due to its complexity. The keyword file has to contain a library that contains all the actions that are to be performed for each specific keyword, and in large systems the number of these keywords can be quite high.

This framework might also not be the best fit when it comes to scalability, which is bad considering requirement REQ-3 in Table 7. As the application scales, and adds more features, new keywords has to be introduced. While maintaining the set of test data in this framework can be easy and fast, maintaining the way keywords work as the application scales can be cumbersome.

When analyzing the Keyword-Driven Testing Framework it is evident that is has a lot of positive sides, with a few negative ones.

The author therefore decided to use this framework as a base when looking at a hybrid version of framework, to see if there were a few combinations of frameworks, or modifications of this framework that could be used to make the Keyword-Driven Testing Framework fit better for `hypso-cli`, while also mitigating a few of the downsides.

5.3.6 Hybrid Testing Framework

As every application is different, the process for testing should be specialized, so it fits within the purpose of the application. In Hybrid Testing Framework, multiple different frameworks are combined, and modified to fit the needs of the application as much as possible, see Section 3.3.6. To this end the author looked at what modifications and combinations could be done with the Keyword-Driven Testing Framework.

The first thing the author looked at was how Keywords were utilized within the framework, as the creation and maintaining of the Keywords file takes quite a bit of time. The first requirement, REQ-0 in Table 7, states that the framework shall imitate normal usage of `hypso-cli`. When using `hypso-cli`, see Section 2.4.2, the user connects to the OPU through the CAN-bus. Once the user is connected, they can then execute commands in the command line interface. Each command is connected to the specific set of code that accomplishes the task that the command represent, in the same way a keyword would. In addition, each command outputs a message after execution, this message differs depending on the command that was executed, and if the command functioned as it should. The user that will create test cases for `hypso-cli`, are HYPSO team members who are already familiar with most of the `hypso-cli` commands.

With this information, the author decided to switch out keywords, when creating test cases, with commands. This results in, when creating test cases, one can not use keywords as a quick way for multiple commands to be entered, instead every command has to be entered. This increases the amount of time it takes when creating test cases, which goes against REQ-1 in Table 7, but it also minimizes the threshold needed for creating tests. The users creating new test cases, do not need to learn what commands each keyword represents. The time it takes when creating test cases with this change, is also still faster than having to create test cases without a test framework.

In the Keyword-Driven Testing Framework example, Figure 21 in Section 3.3.5, there is also an `object` column which indicates what object should be tested on. This is not needed for `hypso-cli`, as every command is tested in the command line interface. This column is therefore not needed, and can be removed.

The resulting test case will then look like the table in Table 8. In this table, each step is a new test case, and inside each test case there can be multiple commands.

Step Nr.	Description	Command
1	Description 1	Command 1
		Command 2
2	Description 2	Command 1
		Command 2

Table 8: Test case in Keyword-Driven Test Framework, with modifications (part 1)

In the table, one can see that the columns `Step Nr.` and `Description` both contain information on the same rows. This indicated that the Step Nr. column can be removed, and the Description column can also be used for indicating a new test case. Additionally, when a command is executed, there is no indication what the test framework should look for to check if the command succeeded or not. In Linear Testing Framework, the user manually records each step of a test case, which can then be automatically executed later, see Section 3.3.1. In doing this the user also acquires the expected result from each command inputted. The same thing can be done in the modified Keyword-Driven Testing Framework. The user can input the commands in `hypso-cli`, and save the output. This output can then be placed into the test case table in a new column.

But so far the modified framework has no implementations of sets of data. To achieve this the author implemented variables that can be inputted with both the commands and expected results from the commands. These variables can then be linked to sets of data contained in data file.

Table 9 displays how a test case would look like with these changes implemented. In the columns `Command` and `Expected Result` There have now been added variables. These variables will connect to sets of data in the data file, and the commands will be executed with all the values found in the data file under the specific variable name.

Description	**Command**	**Expected Result**
Description 1	Command 1 var1	Expected Result 1 Evar1
	Command 2 var2	Expected Result 2 Evar2
Description 2	Command 1 var1	Expected Result 1 Evar1
	Command 2 var2	Expected Result 2 Evar2

Table 9: Test case in Keyword-Driven Test Framework, with modifications (part 2)

These new changes also work well when it comes to maintenance, and when new features are added to `hypso-cli`. When new features are added, previous test cases are not affected, and creating tests with the new features require the same as with old features. I.e. the commands and expected results have to be added to the test case. When it comes to maintenance, if a command in `hypso-cli` is altered, only the test cases that utilize this command have to be changed, the rest will function as normal.

From the result of these modifications to the Keyword-Driven Testing Framework, the new framework will work well with all the requirements found in Table 7.

6 Design & Implementation

Based on the analysis done in Section 5, the author created a design for the test framework. This design is implemented to work with the test menu the author has previously created, see Section 2.5.1, and GitHub Actions, see Section 4.2. The test menu is used for testing multiple different test scripts at once, and GitHub Actions is used for automating testing on target hardware.

6.1 Design

The Design of the entire test framework can be seen in Figure 29. The design revolves around multiple modified Keyword-Driven Testing Frameworks, explained in Section 5, that are used in conjunction with the Test Menu, see Section 2.5.1, and GitHub Actions, see Section 4.3. In the figure different colors are used to differentiate separate parts of the framework.

The first part consists of the 1 up to N amount of blue boxes. Each of these boxes contain the framework explained in the analysis Section 5, with their own Test file (red cylinders in the figure). These test files can contain either one or multiple test cases, and can utilize either the same of different data files (purple cylinders in the figure). Inside each of the blue boxes, there is an orange box. This box represent the test script that is used for extracting information from the test file and the data file, execute the tests on `hypso-cli`, and then compare the results from the commands to the expected results. By utilizing multiple test files in the system, it allows the creation of test files that target selected commands and functionality of `hypso-cli` which gives better overview of the tests, and helps with maintenance when/if some tests need to be altered.

The second and third part of the system consists of the green and red box. The green box represent the Test Menu, see Section 2.5.1, and the red box, see Section 4.3, represent the usage of GitHub Actions. All tests created using the framework are placed in the Test Menu, which allows the creation of a test script that will execute every test after one another, and gives a collected result afterwards. GitHub Actions will use this test script when testing automatically on the target hardware explained in Section 2.5.

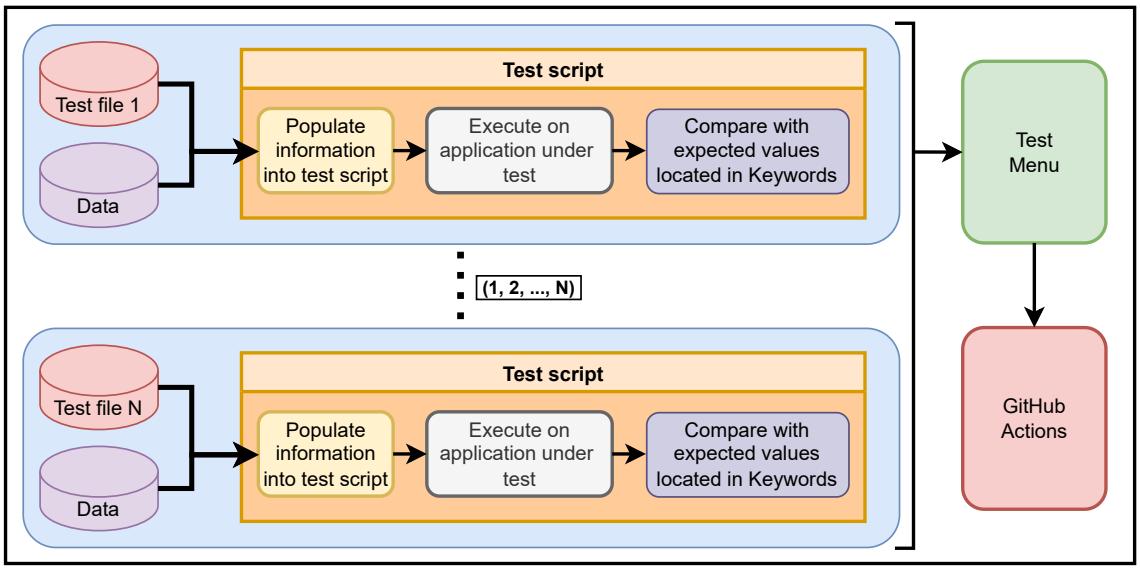


Figure 29: Design flow of testing framework

6.2 Test file and data file

The test file and the data file are the only files that the user of the framework is required to create to be able to test. The contents of these two files are explained in the following sections below.

6.2.1 Data-file

The data file contains sets of data that the commands in the test file can refer to, to be able to test the same commands with different settings. The Data file is set up as a JavaScript Object Notation (JSON) file. JSON is a lightweight data-interchange format, that contains data as text and a syntax derived from JavaScript Object Notation [33].

Code Listing 2 contains an example of a Data file with JSON syntax. In this example there are two main objects, `buffer_settings` and `csp_address`. Both of these objects contain multiple object/value pairs, that can be referenced in the test file. The values each object contains can have the form of a number, a string, or an array of numbers and/or strings. If the value is an array, the command that references the array will be executed as many times as there are elements in the array. For each time the command executes, a new element will be selected. As an example, if the object/value pair `csp_address_id` inside the object `csp_address` (line 9 in Code Listing 2) is referenced. The command that referenced it will execute three times, each time using the next element starting from the first, until all the elements in the object/value pair have been used.

```
1  {
2      "buffer_settings" : {
3          "buffer_csp_address" : 12,
4          "buffer_port" : 25,
5          "buffer_period" : 5,
6          "buffer_file_id" : [32,33,34,35,36,37,38,39,40,41]
7      },
8      "csp_address": {
9          "csp_address_id" : [12,3,3],
10         "csp_address_name" : ["OPU","UHF2","FC"]
11     }
12 }
```

Code Listing 2: Example of Data file with JSON syntax

6.2.2 Test file

The test file contains test cases that define the tests that the framework will execute. The test file is set up as a Comma Separated Values (CSV) file. CSV files are plain text files that contain records of data that are separated by a comma [41]. Since most applications can recognize data separated by comma, importing the data in these files are convenient. As most spreadsheet applications can import CSV files, it makes it very easy to use when creating and editing tests, as each text separated by a comma is inserted into its own spreadsheet cell. When a test file is displayed in this thesis, it will be displayed like a table in a spreadsheet due to readability.

The test file will look similar to Table 9 in Section 5, but with a few more columns that add additional functionality to the framework.

Table 10 and Table 11 contain an example of a test file. The test file has been split in two, due to width of the table and readability, Table 11 is a horizontal continuation of Table 10.

Description	Command	Expected Result
Csp ping with rerun command	csp ping {csp_address,csp_address_id}	Ping received from {csp_address,csp_address_name}
	shell remote oneshot 12 5 find hsi0/temp.txt	hsi0/temp.txt
Csp ping with rerun case	csp ping {csp_address,csp_address_id}	Ping received from {csp_address,csp_address_name}
	shell remote oneshot 12 5 find hsi0/temp.txt	hsi0/temp.txt

Table 10: Example of test file, containing two similar test cases, with different `Rerun case or command` setting (Part 1).

Rerun case or command	End case on fail	Wait time after execution [s]	Timeout [s]
command	No	0	10
none	No	0	10
case	No	0	10
none	No	0	10

Table 11: Example of test file, containing two similar test cases, with different `Rerun case or command` setting (Part 2).

The different columns in the file, with their functionality, are as follows:

- Description
  - Contains the description for the different test cases, as well as being the defining element for when a test case starts. In Table 10 there is a description on every other line between line 2-10. This means that every other line, a new test case is defined.
- Command
  - Contains the commands that are to be tested on `hypso-cli`. These commands can contain variables, that enables the command to be tested with multiple sets data without having to re-write the command. The variables reference specific object/value pairs located in the data file 6.2.1. The variables have to contain the name of the object/value pair written between the symbols "{" and "}". If referencing a nested object, like in line 3 in Table 10 where the command references `csp_address_id` located inside `csp_address`, the variable has to start with the overarching object followed by the object/value pair and separated by a comma symbol. This will then reference the object/value pair located in line 9 in Code Listing 2.
- Expected Result
  - Contains the expected response/result from the terminal after a command has been executed. As different commands in the `Command` column can be executed by the use of variables, the `Expected Result` column also requires the possibility of using variables to check for multiple responses. The variables in the `Expected Result` column, function in the same way as `Command` column.
- Rerun case or command
  - The variables in the `Command` and `Expected Result` columns can be integrated in two different ways, either command-wise or case-wise. Command-wise (chosen by having command in this column) means that the command in the same line as the variable will be re-executed. Where in each execution, either the `Command` and/or the `Expected result` will contain new data from the object/value pair that the variable corresponded to, until all data in the object/value pair has been used. Case-wise (chosen by having case in this column) means that the entire test case will re-execute, each time with new data from the corresponding object/value pair in either in `Command` and/or `Expected Result`, until all data has been used. For example, row 3 in Table 10 has contains a variable that refers to line 9 in Code Listing 2. If line 3 in the `Rerun case or command` column in Table 11 contains command, then the command in row 3 will be executed three times, once with each of the values in the `csp_address_id` object/value pair, before the command in row 4 will execute. If the `Rerun case or`

`command` column contains case, then the entire test case will execute three times, once for each value in the `csp_address_id` object/value pair, before the next test case will execute.

- End case on fail
  - If a command ends up failing, i.e. the result from executing a command does not match the expected result, there are two choices that can be taken.
    Many of the commands that can be executed through hypso-cli require that previous commands have been executed. For example, a command that requires connection to the OPU will fail if the command that turns on the OPU hasn't been executed. By this logic, there might be situations where it would be a waste of time to execute the rest of the test case is an earlier command fails. Adding "yes" to the `End case on fail` column in Table 11, tells the framework to skip the rest of the test case if the command fails.
- Wait time after execution [s]
  - In `hypso-cli`, one might want to wait a set amount of time after executing a command before continuing to the next command. For example, after buffering a file with the `ft buffer file` command, see Section 2.4.2, one might want to wait for the buffering to complete before executing the next command that might check the log file if the buffering succeeded. This is accomplished by adding a number to the `Wait time after execution [s]` column in Table 11. The framework will then wait the set amount of time before continuing. If the `End case on fail` column contains "yes", and the command fails, then the wait time will be skipped as the entire test case will be skipped, and there would be no reason to wait.
- Timeout [s]
  - When a command executes, it might take some time before the output arrives. The number in the `Timeout [s]` column in Table 11 represent the number of seconds that the framework will wait for the expected result to manifest before labeling it as failed. During this time the framework will continuously keep checking for the expected output. If the output manifest during this time it will not continue waiting for the time to run out, before continuing. This might seem very similar to the `Wait time after execution [s]` column. But the `Timeout [s]` will use this time to check for the expected result, while `Wait time after execution [s]` will wait after the expected result has already been checked. This means that if it takes 8 second before a command manifests the expected result, and `Wait time after execution [s]` is set to 20 seconds, it will take 28 seconds before continuing to the next command.

6.3 Test script

This Section goes into more detail on how the test script, seen in Figure 29 functions. The python code created for the test script can be found in Appendix D. Figure 30 displays the system flow of the test script.

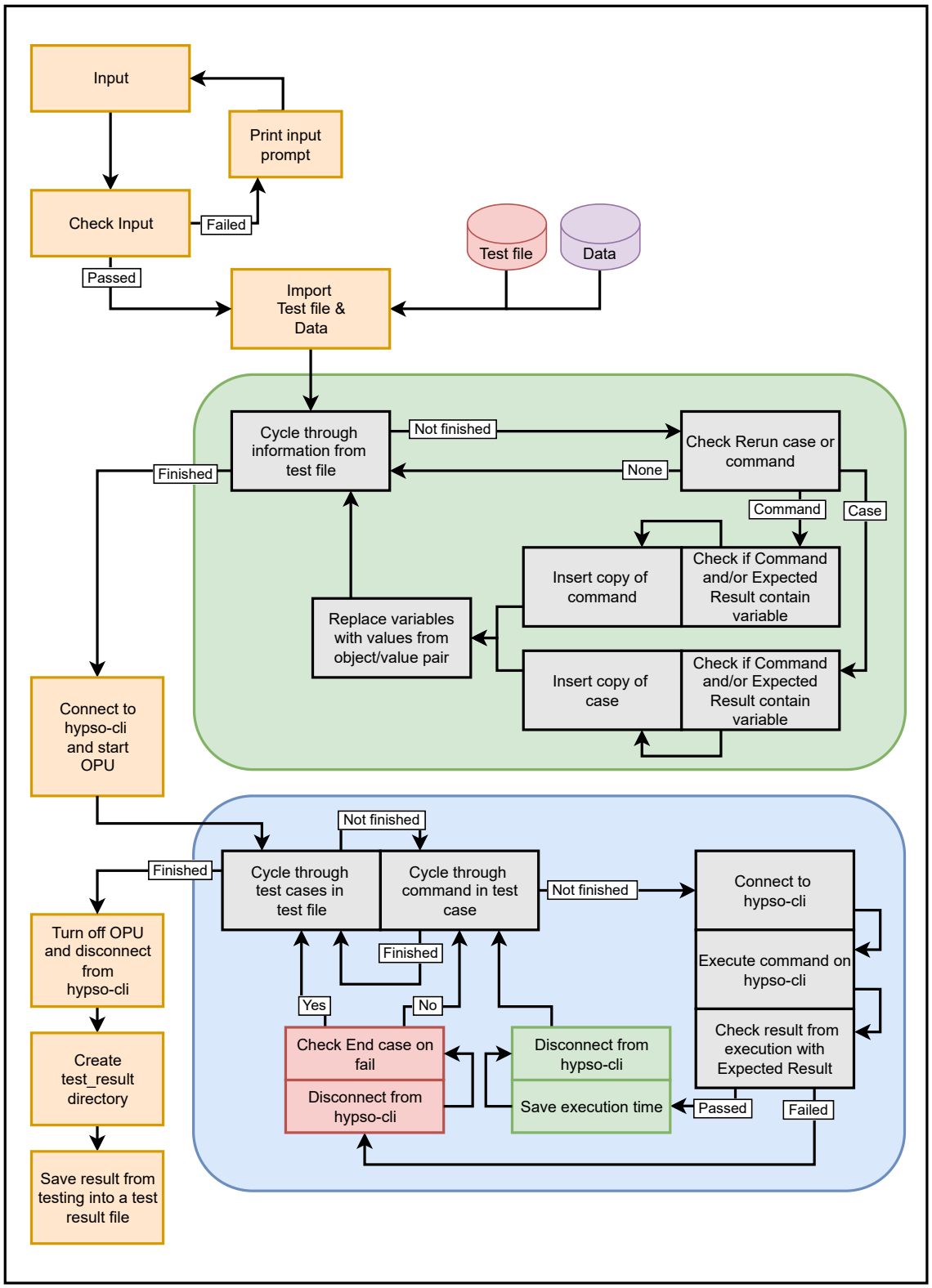


Figure 30: Flowchart of test script used in the modified Keyword-Driven Testing Framework

6.3.1 Input

As seen at the top of Figure 30, the test script for the Modified Keyword-Driven Testing Framework starts with an input. This input has to contain the names of the test file and the data file. If the test file does not contain any variables, then the data file is not needed, and can be omitted. When the test script executes it will also create a log file which will contain everything the test script prints to the terminal. If the user does not feel the need to have a log file, then it can also be omitted. By adding "-n" to the input this is accomplished.

The test script is executed with the following command:

```
./test_script.py
```

In this case, the test script has been executed without any input arguments, which will result in the `Check Input` block in Figure 30 to fail. The script will then print an Input prompt to the terminal, which will look like Figure 31. The `Check Input` block will output the reason for failing in red before the input prompt is printed. In addition, the input prompt explains the usage of the test script as well as a few examples of how it can be executed.

```
Not enough arguments
Usage of automated test file:
 Script to test commands on Lidsat

Input csv test file as first argument
and a json data file as second argument(if needed).
If a log file is not needed -n can be inputted as the first argument
This will then prevent the log file from begin created

Example with data file:
./test_script.py test_file.csv data_file.json

Example without data file:
./test_script test_file.csv

Example for not creating log file:
./test_script -n test_file.csv data_file.json
```

Figure 31: Input prompt when no input arguments have been given at execution of test script for Modified Keyword-Driven Testing Framework

The `Check Input` block in Figure 30 works by first checking the amount of input arguments. It will fail if the amount is less than one or more than three. If a valid amount of input arguments are entered, `Check Input` will check if the files inputted actually exist. As the test script is created to be used on a UNiplexed Information Computing System (UNIX) based system, the inputted files have to contain the relative path to the location of the test script, which can be seen in Figure 11 in Section 2.

The different outputs from the `Check Input` block are as follows:

- Not enough arguments
  - Occurs when the amount of inputs to the test script are fewer than one
- Too many arguments
  - Occurs when the amount of inputs to the test script are higher than 3
- Logging disabled
  - Occurs when one of the inputs to the test script contains "-n".
    In this case the `Check Input` block will not fail, but it will output `Logging Disabled` in yellow so that the user is aware that no log file will be created.

- Inputted test file does not exist
  - Occurs when the amount of input arguments are valid, but the path to the test file relative to the test script is incorrect. In other words, occurs when the test script is unable to find the inputted test file.
- Inputted data file does not exist
  - Occurs when there are two input arguments (three if "-n" for no logging is included), and the path to the data file relative to the test script is incorrect. In other words, occurs when the test script is unable to find the inputted data file.

Once the `Check Input` block has passed, the test script will import the information located in both the test file and the data file. The information in the test file will be placed into a list, while the data file can be used as is with the JSON library for python.

6.3.2 Cycle through information from test file

In Figure 30 the green area are represents when the test script goes through the test and data files and checks for variables, and if there are variables the list containing the information from the test file will be extended according to what is written in the `Rerun case or command` column.

The first block `Cycle through information from test file` checks how many commands exists in the test file, and cycles through all of them one by one. For each cycle it goes to the `Check Rerun case or command` block. The `Rerun case or command` column is required to either contain "command" or "case" to inform how the test script should implement the data from the object/value pair corresponding to the inputted variables. Thus, this is an easy way to check if either the `Command` and/or `Expected Result` contain variables.

Depending on what is written in `Rerun case or command` column, copies of either the case or the command are created in the following blocks. The amount of copies created correlate to the amount of different data located in the object/value pair for each variable. Each of these copies are then furnished with data from the object/value pair, in place of the variable.

Table 12 displays a simplified version of the information located in the list created from the test file before and after `Cycle through information from test file`, with `Rerun case or command` set to "command". Table 12a displays the list before, while Table 12b displays after. In the table after, all the commands are placed after one another, and will be executed as such, before the command `shell remote oneshot 12 5 find hsi0/temp.txt` is executed. This is also the same for variables located in the `Expected Result` column.

Table 13 displays a simplified version of the information located in the list created from the test file before and after `Cycle through information from test file`, with `Rerun case or command` set to "case". Table 13a displays the list before, while Table 13b displays the list after. In the table after, new test cases, that each contain separate data for the command that contained a variable, have been created. These cases have been placed after one another, and will execute as such. This is also the same for variables located in the `Expected Result` column.

The test script will continue with this process, until all the information from the test file has been cycled through, and no more variables are found. At which point it will then exit the green are in Figure 30.

Command	Rerun case or command
csp ping {csp_address,csp_address_id}	command
shell remote oneshot 12 5 find hsi0/temp_file.txt	none

(a) Simplified test file before `Cycle through information from test file`. The variable csp_address,csp_address_name refers to the `csp_address_id` variable in line 9 in Code Listing 2

Command	Rerun case or command
csp ping 12	command
csp ping 3	command
csp ping 3	command
shell remote oneshot 12 5 find hsi0/temp_file.txt	none

(b) Simplified test file after `Cycle through information from test file`

Table 12: Simplified test file before and after `Cycle through information from test file` with `Rerun case or command` set to "command"

Command	Rerun case or command
csp ping {csp_address,csp_address_id}	case
shell remote oneshot 12 5 find hsi0/temp_file.txt	none

(a) Simplified test file before `Cycle through information from test file`. The variable csp_address,csp_address_name refers to the `csp_address_id` variable in line 9 in Code Listing 2

Command	Rerun case or command
csp ping 12	case
shell remote oneshot 12 5 find hsi0/temp_file.txt	none
csp ping 3	case
shell remote oneshot 12 5 find hsi0/temp_file.txt	none
csp ping 3	case
shell remote oneshot 12 5 find hsi0/temp_file.txt	none

(b) Simplified test file after `Cycle through information from test file`

Table 13: Simplified test file before and after `Cycle through information from test file` with `Rerun case or command` set to "case"

6.3.3 Connect to hypso-cli and start OPU

After the list created from the information from the test file has been extended with sets of data instead of variables, it is now ready to be executed on `hypso-cli`. Before testing though, the test script tries to start `hypso-cli`. The `hypso-cli` that the test script utilizes is the one located in the `assembly-integration-test` repository, see Section 2.4.3. This `hypso-cli` has been created by the `hypso-sw` repository, see Section 2.4.1, and then moved into the `assembly-integration-test` repository. If the test script is able to start `hypso-cli`, it will then input a command that tells the EPS to provide power to the OPU. Most of the commands and functionalities on the HYPSO satellites require the OPU. Therefore instead of always having to start the OPU in every test, this happens automatically. The OPU will turn itself off after a set amount of time if it is not used, and when the test script finishes it will also try to turn of the OPU. Therefore, starting it preemptively is not a major concern, even if the test cases in the test file might not use OPU.

6.3.4 Cycle through test cases in test file

After the test script has made sure it is able to connect to `hypso-cli`, and has started the OPU, it will go forward to the blue box in Figure 30. This box represent the actual testing of the list of test cases created from the previous steps. The testing process starts by cycling through the test cases, where in each cycle it will go through the commands in each test case. For each command, a python subprocess is started that starts `hypso-cli`, and inputs the command. A python subprocess is a module that allow the creation of new processes that can be used for executing external code and programs [42], in this case `hypso-cli`.

A timeout handler is also started at the same time, this timeout handler will wait for the set amount of time located in the `Timeout [s]` column for the specific command, before it triggers and marks the command as failed. While the timeout handler is ongoing, the test script continuously check the output from `hypso-cli`. If `hypso-cli` returns the same information that is stored in `Expected Result` column for the specific command, then the timeout handler is preemptively

stopped, and the command is marked as passed. This takes place in the `Check result from execution with expected result` block. In addition, if the command passed, then the execution time, the time from inputting the command into `hypso-cli` and until it got an output that matched the expected results, is saved. If the command failed, an execution time will not be saved. No matter if the command passed or failed, the connection to `hypso-cli` through the python subprocess is terminated. If the command passed, the test script continues cycling through the commands in the test case until finished. If the command failed, the `End case on fail` column for the specific command is checked. A "yes" in this column tell the test script to skip the rest of the commands in the test case. These commands are then also marked as "skipped" instead of either passed of failed.

Once the test script has either finished cycling through all commands in a test case, or skipped a test case, it starts the testing process over again for the following test cases in the test list, until all test cases have been cycled through. At which point it will exit the blue box in Figure 30.

6.3.5 Disconnect from hypso-cli

Once the test script has finished cycling through the test cases, it will turn off the OPU, and disconnect the first connection it made to hypso-cli. If the OPU is already off, turning it off again will not return an error as the command for turning on/off the OPU works by sending a state change for the OPU connection to the EPS. Turning on the OPU occurs by sending the state "1", while turning off is sending the state "0". If the OPU is turned off before this point of the test script flow in Figure 30. The state change in the EPS is changed from "0" to "0", which will not make any difference.

6.3.6 Save results from test script into test result file

After disconnecting from `hypso-cli`, the test script will create a result file. The test script will create a `test_result` directory that will be located inside the `automated_test` directory, see Figure 11 in Section 2.4.3. If this directory already exists, a new one will not be created.

The test scrip will then create a result file that will be placed inside the test_result directory. The name of the result file will have a specific syntax:

```
(test file name)_(date)T(time)
```

The values inside the parenthesis(including the parenthesis) will be switched out with the name of the test file inputted as an argument when executing the test script, the date of the day the test script was executed, and the time of execution respectively. For example, if the test was named `Test1` and the testing started 7. May 2022, at 04:30:00 UTC, the result file would be named `Test1_220507T043000`

The result file is a CSV file for the same reasons the test file is a CSV file, see Section 6.2.2. An example of a result file that has been created can be seen in Table 14 and Table 15. The test file that used when creating this result file contained one test case, with two commands, where none of the commands contained variables. The result file contains the same info as in the test file, with some additional information. At the top of the result file the date the testing took placed is displayed, as well as the Git branch and Git commit the `hypso-sw` repository had when the `hypso-cli` used was created. A summary detailing how many tests were passed, failed, and skipped, with their overall percentage is also shown.

In the result file the UTC time for when a command was executed, as well as if it passed, failed, or skipped is shown. If the command passed, then the time it took to execute is shown, if the command either failed or was skipped, the execution time for that line will contain `NA`. A more thorough look on the result file is shown in Section 7.

Date	2022-05-07		Tests	Amount	Percentage
Git branch	HEAD		Passed	2	100.0%
Git commit	3efd1e1		Failed	0	0.0%
			Skipped	0	0.0%
Time[UTC]	Test Results	Execution time[s]	Description	Command	Expected Result
18:48:23:670607	Passed	0.0353543758392334	Check csp ping with command rerun	csp ping 12	Ping received from OPU
18:48:23.708550	Passed	0.05288124084472656		shell remote oneshot 12 5 find hsi0/temp.txt	hsi0/temp.txt

Table 14: Example of result file, containing one test case with two commands, none of which contain variables (Part 1).

Rerun case or command	End case on fail	Wait time after execution[s]	Timeout[s]
None	No	0	10
None	No	0	10

Table 15: Example of result file, containing one test case with two commands, none of which contain variables (Part 2).

6.4 Test menu

The Test Menu system is used for creating a larger test script that can be used to execute multiple test files sequentially, and give a collective result afterwards, see Section 2.5.1. Figure 32 shows a snippet of the menu system in use. There are two tests that have been selected for testing, both of these tests uses the test script for the framework in this thesis, seen under the column called `Tests`. `Arg1` and `Arg2` respectively contain the first and second argument that are to be used with the test script. The first test will use a test file named `test1.csv`, and the second test will use a test file named `test2.csv`. Both of these test files uses the same data file containing object/value pairs, namely `data.json`.

```
  Selected for test   | Tests              | Arg1             | Arg2             | Arg3              |
  ------------------------------------------------------------------------------------------------
3 YES                 | test_script.py     | test1.csv        | data.json        |
  YES                 | test_script.py     | test2.csv        | data.json        |
  NO                  | test1.py           | folder_name      | mode             | 1                 |
  NO                  | test2.py           |                  |                  |                   |
  ------------------------------------------------------------------------------------------------
```

Figure 32: Snippet taken from test menu, containing 4 tests where two have been selected for testing.

Figure 33 contains a second snippet from the test menu. Here one can see a choice of generating a new `auto_test_sequence.csv` file. This file can will be used in conjunction with the `auto_test_execution.py` python executable to execute multiple tests automatically.
Figure 34 displays two possible outcomes from the `auto_test_execution.py` executable. Figure 34a displays the outcome when both tests pass, while Figure 34b shown when both tests fail. Note that when a test fail, both the name of the test script and the input arguments are outputted as under Failed Tests. This is quite advantageous when using GitHub Actions, and will be more explained in Section 7.

```
Navigation of menu
Quit: q or Q
Enter: enter key

YES and NO are toggled values


Change default values
Press D or d

Generatge new test_sequence.csv file
Press G or g

Generatge new auto_test_sequence.csv file
Press A or a

Execute tests in test_sequence.csv file
Press R or r
```

Figure 33: Snippet from test menu explaining the controls and options for the menu

```
Testing Finished

2 of 2 Tests Passed
Pass rate: 100.0%
```

(a) Example of output from `auto_test_execution.py` when two tests both pass.

```
Testing Finished
Failed Tests:
 test_script.py test1.csv data.json
 test_script.py test2.csv data.json

0 of 2 Tests Passed
Pass rate: 0.0%
```

(b) Example of output from `auto_test_execution.py` when two tests both fail.

Figure 34: Two possible outcomes from the `auto_test_execution.py` executable. One where two tests pass, and one where two tests fail. When a test fail, the name of the test script and the input arguments are outputted under Failed Tests.

6.5 GitHub-Actions

To automatically test new test files and/or automatically test previously created test files at certain times, on target hardware, GitHub Actions is utilized. GitHub Actions will be used for executing the `auto_test_sequence.py` python executable at specific trigger events, see Section 4.3. Before this can take place, GitHub Actions has to be set up to work with the HYPSO test setup specified in Section 2.5. The following sections explain the specifics in setting up GitHub Actions.

6.5.1 Setting up GitHub Action Runner

The GitHub Actions Runner is the application used for executing the testing. Since this testing is to take place on target hardware, the runner is required to be self-hosted, see Section 4.3. To give the self-hosted runner access to target hardware, it has to be set up on the Operational Computer on the HYPSO test setup, see Figure 13 in Section 2.5.

When setting up this self-hosted server, a guide made by GitHub was used [17]. Through the use of the guide, the runner was automatically connected to the `hypso-sw` repository, and could be used to execute workflow files located in the `.github/workflows` directory in the repository.

Additionally, the GitHub Actions runner required to be able to execute python files, as the executable used for executing the `auto_test_sequence.csv` file created from the test menu is a python file; `auto_test_execution.py`. To enable this, an external Action called `setup-python`, was used [16]. This Action provides a guide for downloading, installing and adding to PATH an available version of Python in the tools cache on the test setup that GitHub Actions is able to utilize.

6.5.2 Workflow files

For GitHub Actions, two workflow files have been created. One in the `hypso-sw` repository and one in the `assembly-integration-test` repository. Both of these repositories will trigger testing when a GitHub pull request, see Section 4.2, is issued to the master branch. For the `assembly-integration-test` repository, a pull request to the master branch might entail new tests that are to be implemented into the testing. Having the test setup automatically test when new tests are added can be used to ensure the tests work with previous confirmed version of `hypso-cli`. For the `hypso-sw` repository, a pull request to the master branch might entail new functionality to the on-board software and/or `hypso-cli`, these new functionalities have to be tested to ensure they are working as intended, and do not introduce bugs.

The workflow file in `hypso-sw` will execute the testing on the self-hosted runner that is set up on

the HYPSO test setup, while the workflow file in `assembly-integration-test` will execute on a GitHub hosted runner, and will send a dispatch message to the workflow file in `hypso-sw`, which will cause the workflow file in `hypso-sw` to execute. Figure 35 displays the workflow of both files while Code Listing 3 and Code Listing 4 display the YAML code for both files respectively. At the start of both code listings, one can see the events that will trigger GitHub Actions for the files. For `hypso-sw`, the workflow file will trigger on pull requests to the master branch, and to workflow dispatches. A workflow dispatch is an event that can be triggered through the use of the GitHub REST API [19]. The workflow file in `assembly-integration-test` uses an external Action called `github-script`, which provides an easy way to use the `GitHub REST API` through workflow files [43], to send a dispatch event to the `hypso-sw` workflow file. Additionally, as the workflow in `assembly-integration-test` does not require to be connected to the test setup, it can be executed on GitHub hosted servers, which can be seen on line 9 in the workflow file.

In Figure 35 each block inside the green and blue blocks represent different steps in the workflow files. As steps execute sequentially, the arrows in the figure represents the order they execute.

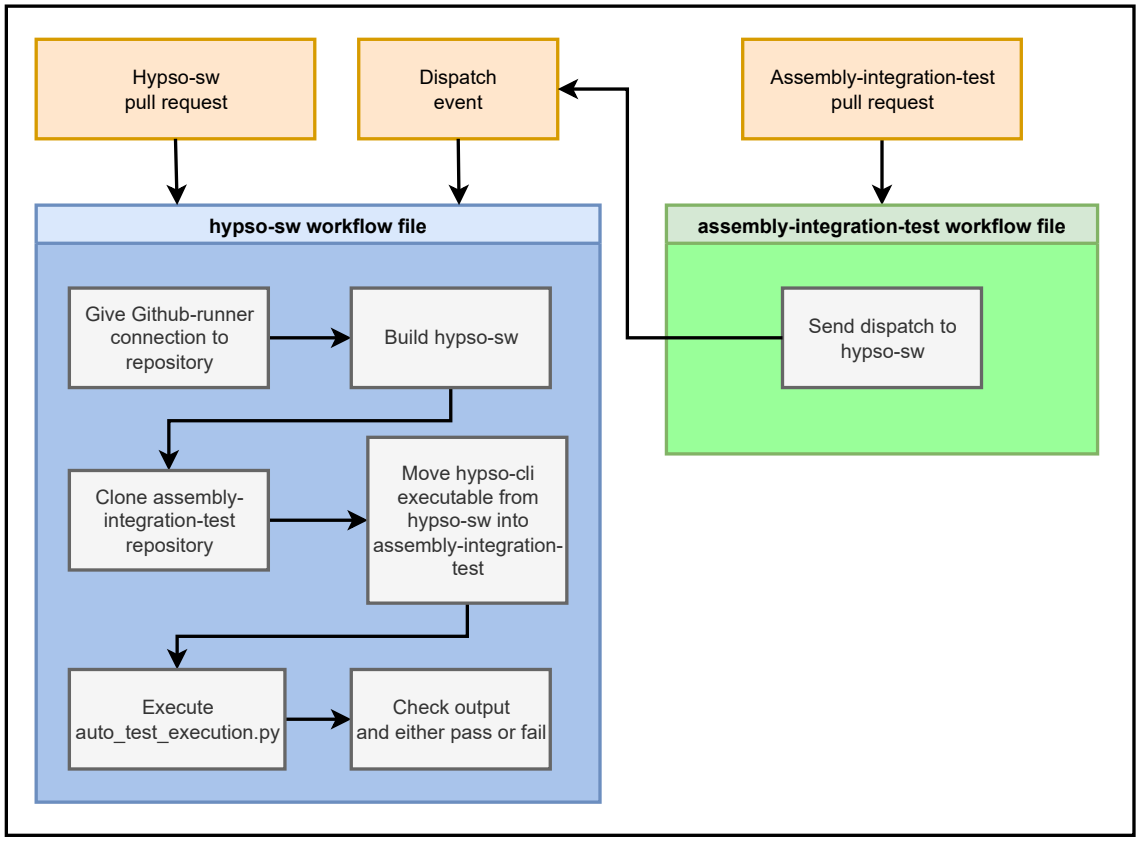


Figure 35: Flowchart of GitHub Actions workflow files for `hypso-sw` repository and `assembly-integration-test` repository.

```yaml
name: Workflow for connecting to assembly-integration-test-repo
on:
  pull_request:
    branches: [master]
  workflow_dispatch:

jobs:
  Testing:
    runs-on: [self-hosted, lidsat]
    name: A job to do CI tests
    steps:
      - name: Checkout branch
        uses: actions/checkout@v2
        with:
          submodules: recursive
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}

      - name: Building
        uses: NTNU-SmallSat-Lab/hypso-sw-build-check@main
        env:
          GITHUB_TOKEN: ${{ secrets.G_ACCESS_TOKEN }}

      - name: clone assembly-integration-test
        uses: actions/checkout@v2
        with:
          repository: NTNU-SmallSat-Lab/assembly-integration-test
          token: ${{ secrets.G_ACCESS_TOKEN }}
          path: assembly-integration-test
          ref: master

      - name: move hypso-cli executable
        run: cp build/x86/hypso-cli assembly-integration-test/

      - name: Run script
        uses: mathiasvr/command-output@v1
        id: run_script
        with:
          run: ./assembly-integration-test/test_menu/auto_test_execution.py
        env:
          key: ${{ secrets.key }}

      - name: Check if all tests passed
        if: |
              false == contains(steps.run_script.outputs.stdout, 'Pass rate: 100.0%')
        uses: actions/github-script@v3
        with:
              script: |
                    core.setFailed('All tests did not pass!')
```

Code Listing 3: Code for workflow file in hypso-sw repository

```yaml
name: Dispatch to hypso-sw
on:
  push:
    branches: [master]
  pull_request:
    branches: [master]
jobs:
  dispatch:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/github-script@v6
        with:
          github-token: ${{ secrets.G_ACCESS_TOKEN }}
          script: |
            const result = await github.rest.actions.
               createWorkflowDispatch({
              owner: 'NTNU-SmallSat-Lab',
              repo: 'hypso-sw',
              workflow_id: 'test_framework.yml',
              ref: 'test-framework-github-actions'
            })
```

Code Listing 4: Code for workflow file in assmebly-integration-test repository

7 Results

This section contains results from three tests executed on the entire framework. Two simple tests used for testing that every function of the framework is operational, and one larger, and more complex test which represent a more accurate test to what might be executed with the test framework. The tests have all been placed inside the test menu in Figure 36, and a new `auto_test_sequence.csv` file has been created for every run of the test framework.

The two simple tests, both use the same test file, but different data files. The first data file contains no errors, while the second data file contains an intended error. The more complex test, contain its own test file, and data file.

In all tests, pull requests in both `assembly-integration-test` and `hypso-sw` repositories are tested.

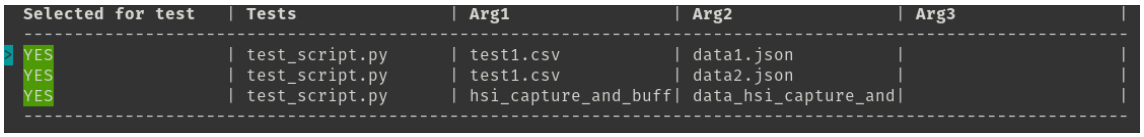


Figure 36: Tests located inside test menu

7.1 Simple Test File

Figure 16 and 17 displays the test file used for the testing of the simple tests. The file contains two test cases, the first one which will has "command" in the `Rerun case or command` column, while the second test case has "case". Both test cases have two commands, with variables in their first command and expected result, and no variable in the second. The object/value pair these variables corresponds to are displayed in the data files which are shown in Section 7.2.1 and Section 7.3.1.

The command `csp ping` tries to send a ping from the Operational Computer to another component on the HYPSO test setup, and listens for an answer. The output on hypso-cli, if successful, contains `Ping received from` in addition to the letters from the component it tried to ping. The command `shell remote oneshot 12 5 find config` uses the shell command `shell remote oneshot` to connect to the OPU, execute a command once, and then disconnect. The command executed on the OPU is `find config`. This command checks the memory on the OPU if it contains a file called `config`. The output from hypso-cli, if successful, contains `config`.

Lastly the second command in the first test case, contains the number 50 in `Wait time after execution [s]` column, which indicated that the framework should wait 50 second after this command has been executed before going to the next.

Description	Command	Expected Result
Csp ping with rerun command	csp ping {csp_address,csp_address_id}	Ping received from {csp_address,csp_address_name}
	shell remote oneshot 12 5 find config	config
Csp ping with rerun case	csp ping {csp_address,csp_address_id}	Ping received from {csp_address,csp_address_name}
	shell remote oneshot 12 5 find config	config

Table 16: Test file used in testing of simple tests, containing two test cases (Part 1).

Rerun case or command	End case on fail	Wait time after execution [s]	Timeout [s]
command	Yes	0	10
none	Yes	50	10
case	No	0	10
none	Yes	0	10

Table 17: Test file used in testing of simple tests, containing two test cases (Part 2).

7.2 Simple first test

In the first testing of the framework, a pull request in both `assembly-integration-test` and `hypso-sw` repositories are executed. Both of these pull request should execute the hypso-sw workflow file which executes the test script explained in section 6.3 with a test file, and a data file which contains no errors.

7.2.1 First Data File

Code Listing 5 displays the data file used in tandem with the test file. This file contains two object/value pairs that correspond to the variables used in the test file. `Csp_address_id` contains two IDs that refer to the components OPU and EPS in HYPSO test system, while `csp_address_name` contains the names of the components.

```
1 {
2     "csp_address": {
3         "csp_address_id" : [12,4],
4         "csp_address_name" : ["OPU","EPS"]
5     }
6 }
```

Code Listing 5: Data file used when testing, containing two object/value pairs.

7.2.2 Pull request assembly-integration-test

When a pull request is issued in the `assembly-integration-test` repository, a merge message comes up. The merge message displays whether the workflow files have passed or failed. The merge message for the for this pull request passed, which can be seen in Figure 37. This indicates that the workflow file in assembly-integration-test, see Code Listing 4 in Section 6.5, that sends a dispatch event to the workflow file in hypso-sw worked as intended. On the GitHub website, in the hypso-sw repository, one can see the GitHub Actions runs that have been executed. Here one can see that the workflow file in hypso-sw has been executed, and that it was triggered due to a dispatch event, as seen in Figure 38. The text `A job to do CI tests` relate to the name seen at line 10 in Code Listing 3, and the green checkmark indicates that it was successful.

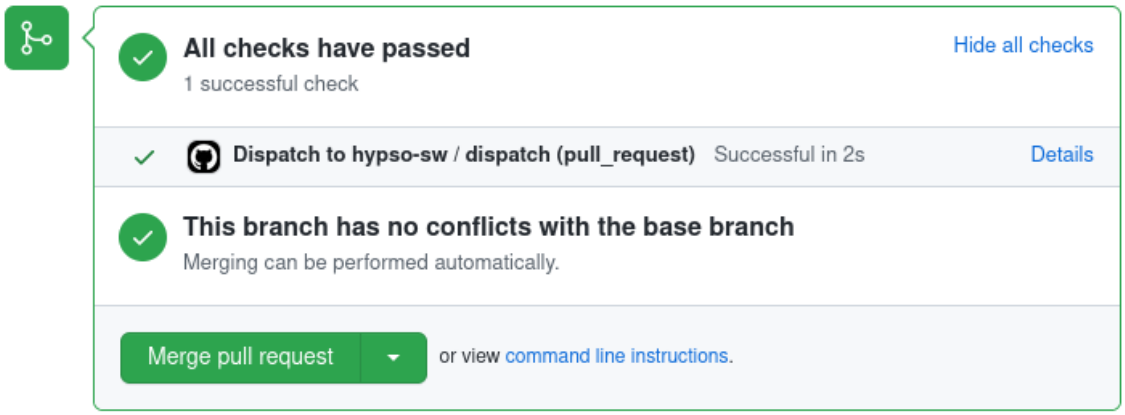


Figure 37: Merge message in `assembly-integration-test` repository when issuing a pull request. In this merge message the Dispatch workflow file has executed successfully.

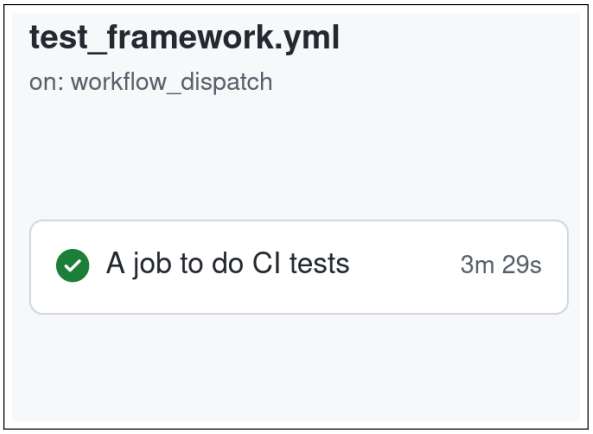


Figure 38: Result from a GitHub Actions run in the `hypso-sw` repository that has been triggered by a dispatch event.

7.2.3 Pull request hypso-sw

When a pull request is issued in the `hypso-sw` repository, a merge message also comes up here, which can be seen in Figure 39. The merge message here indicates that the workflow file in hypso-sw, see Code Listing 3 in Section 6.5, has been executed and that it was successful. The merge message also displays the name of the workflow which has been executed, and this name is the same as line 1 in the Code Listing for the workflow file. Additionally, in Figure 40 one can see that this time the workflow was triggered by a pull request instead of a dispatch event.

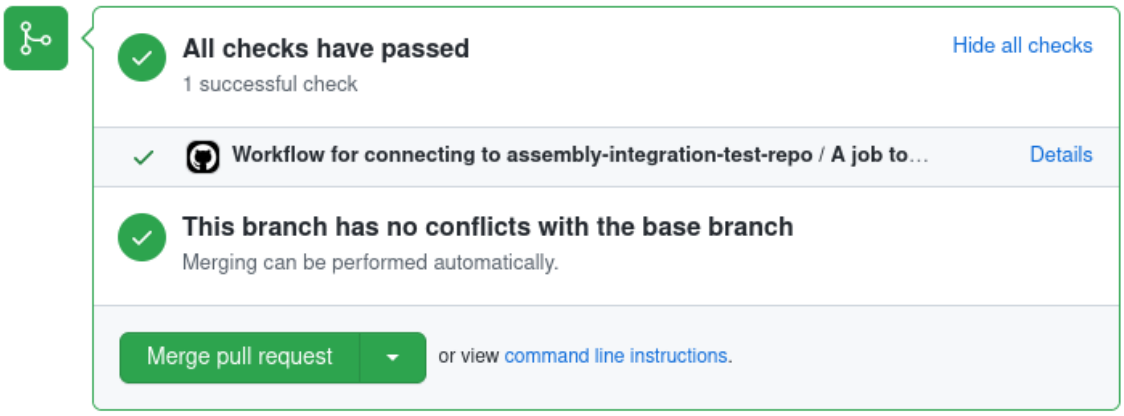


Figure 39: Merge message in `hypso-sw` repository when issuing a pull request. In this merge message the workflow file has executed successfully.

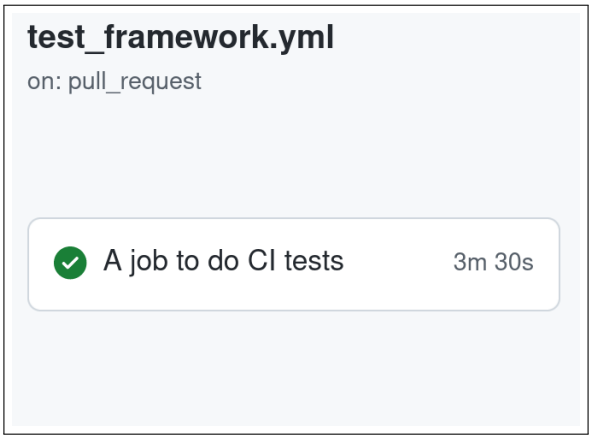


Figure 40: Result from a GitHub Actions run in the `hypso-sw` repository that has been triggered by a pull request.

7.2.4 First test result file

The result file created from the first testing can be seen in Table 18 and Table 19. Here one can see that, from the test file and the data file used, seven commands were executed. The output from every command matched what was written in the `Expected Result` column, which means that every command passed. Additionally, as the first `shell remote oneshot 12 5 find config` command in the test file had 50 in the `Wait time after execution [s]` column, the framework should wait 50 second after this command before continuing to the next. Under the `Time[UTC]` column in the result file one can see that this wait occurred, as the time of execution the first `shell remote oneshot 12 5 find config` had is 50 seconds before the time of execution for the following command (including some milliseconds as wait only occurs after the command has passed, and some internal processing time that the test script uses).

Date	2022-05-23		Tests	Amount	Percentage
Git branch	HEAD		Passed	7	100.0%
Git commit	3efd1e1		Failed	0	0.0%
			Skipped	0	0.0%
Time[UTC]	Test Results	Execution time[s]	Description	Command	Expected Result
00:42:35.094229	Passed	0.0353543758392334	Check csp ping with rerun case	csp ping 12	Ping received from OPU
00:42:35.129741	Passed	0.11270570755004883		csp ping 4	Ping received from EPS
00:42:35.245080	Passed	0.050455331802368164		shell remote oneshot 12 5 find config	config
00:43:25.348682	Passed	0.032892465591430664	Check csp ping with rerun case	csp ping 12	Ping received from OPU
00:43:25.384207	Passed	0.048543691635131836		shell remote oneshot 12 5 find config	config
00:43:25.435474	Passed	0.11247801780700684	Check csp ping with rerun case	csp ping 4	Ping received from EPS
00:43:25.550533	Passed	0.049387454986572266		shell remote oneshot 12 5 find config	config

Table 18: Result file from first simple test, where all commands passed (Part 1).

Rerun case or command	End case on fail	Wait time after execution[s]	Timeout[s]
command	Yes	0	10
command	Yes	0	10
none	Yes	50	10
case	No	0	10
none	Yes	0	10
case	No	0	10
none	Yes	0	10

Table 19: Result file from first simple test, where all commands passed (Part 2).

7.3 Simple second test

The second test of the framework will test the same way as the first test, but instead of only testing with the test file once with a correct data file, this testing also includes testing the test file with a data file that contains intentional errors.

7.3.1 Second Data File

The second data file can be seen in Code Listing 6. This file is the same as the first test file, but this time it contains an intended error in the form of a 0 in line 3, which is marked in green. In the first data file, see Code Listing 5, which passed the testing with no errors, contains the number 4. The number 0 does therefore not match the ID of the EPS.

```
1 {
2     "csp_address": {
3         "csp_address_id" : [12,0],
4         "csp_address_name" : ["OPU","EPS"]
5     }
6 }
```

Code Listing 6: Data file used when testing, containing two object/value pairs with one inteded error marked in green.

7.3.2 Pull request assembly-integration-test

When a pull request now occurs in the `assembly-integration-test` repository, the workflow file in the merge message is still marked as successful, as seen in Figure 41. This is due to a shortcoming of the GitHub REST API, where it will send a dispatch event, and as soon as that dispatch event is received, it will be marked as passed. This shortcoming is discussed further in

Section 8 and Section 10. The workflow file in `hypso-sw` is triggered by the dispatch event as seen in Figure 42, but in this case it has been marked as failed, as seen by the red cross. On the GitHub website, one can then look further into the steps executed by the workflow, Figure 43 contains a snippet from the website and displays why the test failed. The `Run script` stage contains the output from the `auto_test_execution.py` python executable, see Section 2.5.1, and here one can see that the `test_script.py` executable with the test file and the second data file failed. As this test failed, the pass rate of the testing did not match 100%, which is what the `Check if all tests passed` step looks for, see line 45 in Code Listing 3.

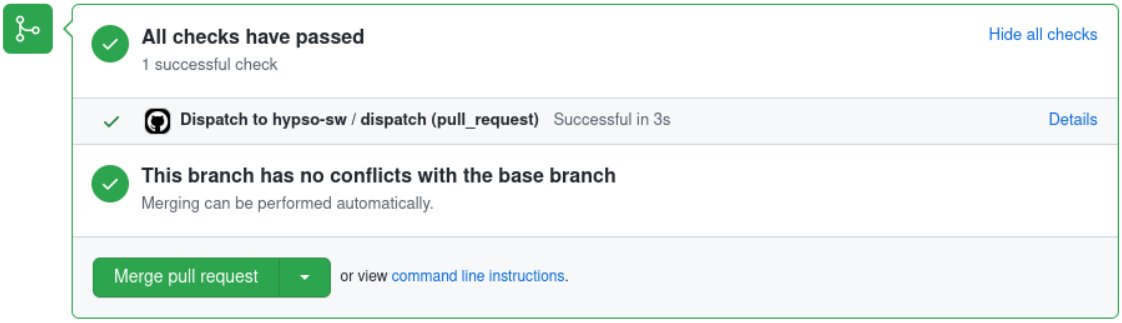


Figure 41: Merge message in `assembly-integration-test` repository when issuing a pull request. In this merge message the workflow file has executed successfully.

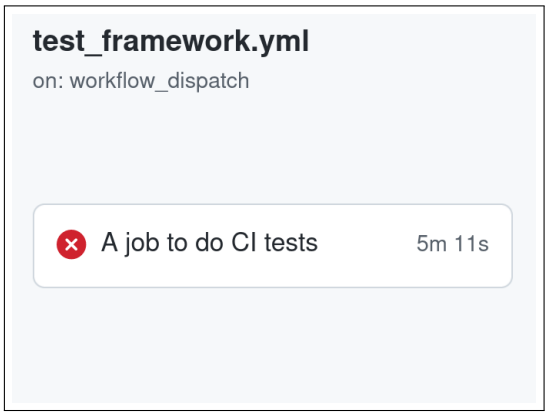


Figure 42: Result from a GitHub Actions run in the `hypso-sw` repository that has been triggered by a dispatch event. The GitHub Actions run was not successful.

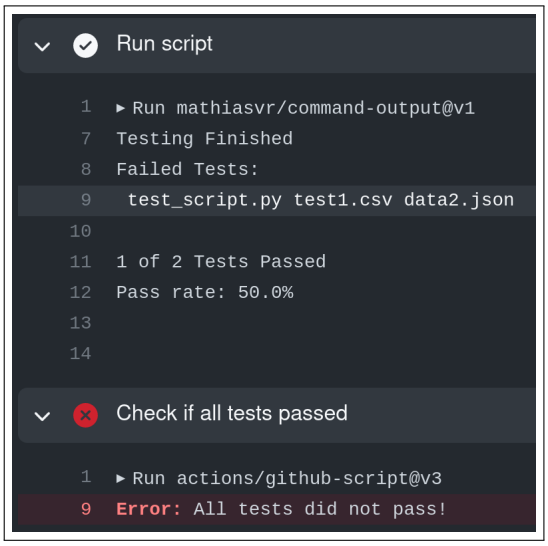


Figure 43: Result from a failed GitHub Actions run, where the test `test_script.py test1.csv data2.json` failed.

7.3.3 Pull request hypso-sw

A pull request in the `hypso-sw`, on the other hand, does produce an error in the merge message, see Figure 44. In this figure one can see that the workflow file in `hypso-sw` did not pass, which matches with Figure 45, which displays that workflow was triggered by a pull request, and Figure 46, which displays why the test failed.

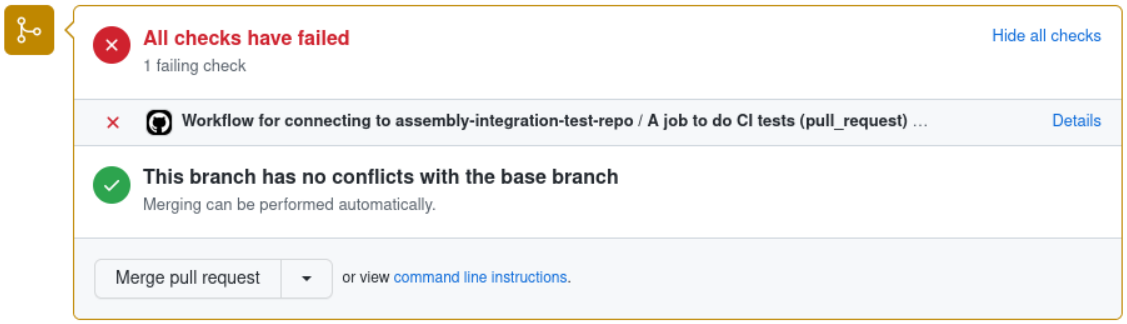


Figure 44: Merge message in `hypso-sw` repository when issuing a pull request. In this merge message the workflow file has not executed successfully.

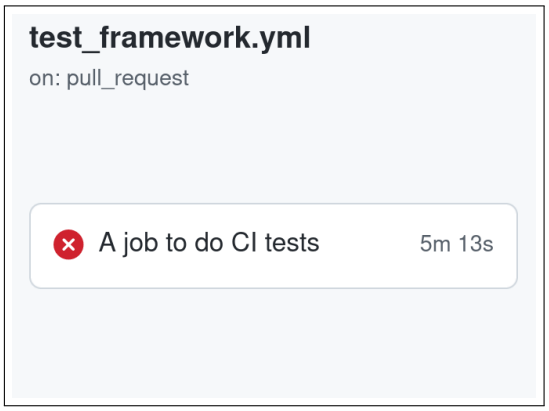


Figure 45: Result from a GitHub Actions run in the `hypso-sw` repository that has been triggered by a pull request. The GitHub Actions run was not successful.

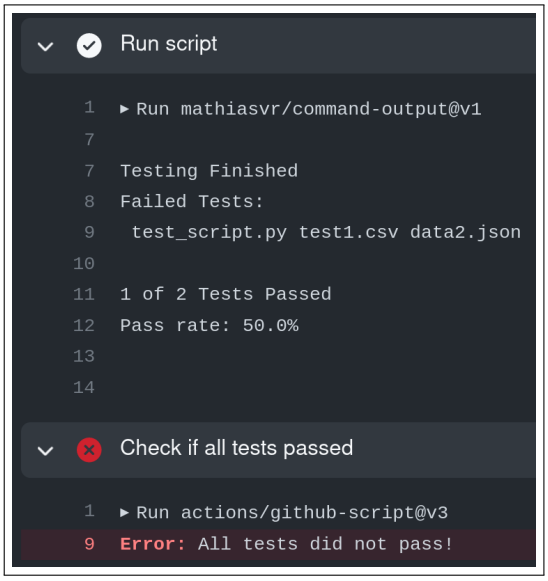


Figure 46: Result from a failed GitHub Actions run, where the test `test_script.py test1.csv data2.json` failed.

7.3.4 Second test result file

Executing the second testing resulted in two test result files, where the first test result file is identical(albeit with different time that the commands were executed) to the test result from the last tests as they used the same test file and data file. This test result file is therefore omitted here.

Table 20 and Table 21 display the result file from the execution of the `test_script.py` file with the test file and the second data file.

In the result file, one can see that 4 commands passed, 2 failed, and 1 was skipped. The time of execution of these commands can be seen in the `Time[UTC]` column, where the command that was skipped contains NA (not applicable) instead of a time, since it was never executed. The commands that either failed and skipped both contain NA in the `Execution time[s]` column as well. The commands that failed are the commands that sent `csp ping 0` and expected to get an output saying they received an answer from the EPS, which coincides with the intended error in the data file. Additionally, the skipped command occurred due to a failed command that happened before it in the test case, which contained "Yes" in the `End case on fail`. As the skipped command was not executed, the framework did not wait the 50 seconds that the command has in the `Wait time after execution[s]` column. All the commands contained 10 in the `Timeout[s]` column,

indicating that the framework would wait 10 seconds for the expected result before continuing. By examining the `Time[UTC]` column one can see that the commands following a failed command occurred approximately 10 second later (approximation due to internal processing time of the test script).

Date	**2022-05-23**		**Tests**	**Amount**	**Percentage**
Git branch	HEAD		Passed	4	57.14285714285714%
Git commit	3efd1e1		Failed	2	28.571428571428571%
			Skipped	1	14.285714285714285%
Time[UTC]	Test Results	Execution time[s]	Description	Command	Expected Result
12:44:53.589105	Passed	0.0332493782043457	Check csp ping with rerun command	csp ping 12	Ping received from OPU
12:44:53.625463	Failed	NA		csp ping 0	Ping received from EPS
NA	Skipped	NA		shell remote oneshot 12 5 find config	config
12:45:03.627976	Passed	0.03315258026123047	Check csp ping with rerun case	csp ping 12	Ping received from OPU
12:45:03.662982	Passed	0.05002140998840332		shell remote oneshot 12 5 find config	config
12:45:03.715067	Failed	NA	Check csp ping with rerun case	csp ping 0	Ping received from EPS
12:45:13.717974	Passed	0.04864645004272461		shell remote oneshot 12 5 find config	config

Table 20: Result file from second test, where 4 commands passed, 2 failed, and 1 was skipped. (Part 1).

Rerun case or command	End case on fail	Wait time after execution[s]	Timeout[s]
command	Yes	0	10
command	Yes	0	10
none	Yes	50	10
case	No	0	10
none	Yes	0	10
case	No	0	10
none	Yes	0	10

Table 21: Result file from second test, where 4 commands passed, 2 failed, and 1 was skipped. (Part 2).

7.4 Complex test

The test in this section will test that the framework is able to handle tests that are more faithful to the tests that the HYPSO team might use. This test is more complex than the previous two tests, as it tests functionality of commands that utilizes multiple components of the HYPSO test setup, see Section 2.5. This test will take a HSI image, split it up into chunks, and store the chunks in a buffer file on the PC, which needed for transferring pictures to the Operational Computer from the satellite [31].

7.4.1 Test file & Data file

The test file can be seen in Table 22 and Table 23. Some commands in the test file do not have an expected result, this is due to `hypso-cli` not giving an output when these commands are executed. The commands in this test file also contain much more variable timings than the

The test starts with the first test case, which removes the folder with the same name that will be created by the capture of the HSI image, on the On-board Processing Unit (OPU). The second test case will turn on the HSI imager, followed by acquiring its temperature. When the command for turning on the HSI imager is sent, `hypso-cli` outputs OK. To ensure it is actually turned on, one can check its temperature. If the temperature is returned then the HSI imager is tuned on. Afterwards, a HSI image containing 250 frames (-n 250 in the command) is captured, the captured image is then compressed through software (-s in the command), before it is saved on the OPU as `compressed_cube.bip.cmpr`.

Taking a HSI image uses some time depending on the amount of frames. Therefore, the capture has a timeout of 200 seconds, with 20 seconds wait time after execution for software compression. In some cases, this wait time might be long enough for the OPU to turn off, thus, a command to turn it back on, and a ping to check connection is also sent. The `compressed_cube.bip.cmpr` is then split into several 8 MB chunks depending on the size of the image. These chunks are then ensured they are placed on the OPU.

The third test case clears the buffer files on the PC, to make them ready for buffering the new chunk files. Lastly, the last test cased buffers the chunk files to the OPU, and checks the log for the On-board Processing Unit (OPU) to see that the buffering has both started, and completed for all the files.

Description	Command	Expected Result
Clear hsi folder on the OPU	shell remote oneshot 12 5 rm -r hsi{hsi_folder}	
Turn of hsi camera and capture an image	shell remote oneshot 4 5 output 8 1 0	OK
	hsi gettemp	Temperature is
	hsi capture -n 250 -s	created folder: hsi{hsi_folder}
	shell remote oneshot 4 5 output 10 1 0	OK
	csp ping 12	Ping received from OPU
	hsi chunkify {hsi_folder} 8 compressed_cube.bip.cmpr	Found file, starting partitioning
	opu list	hsi{hsi_folder}/compressed_cube.bip.cmpr_chunk{chunk_amount}
	ft download cancel 6	
Request the buffer file on the PC to be cleared	ft clear 6 {buffer,buffer_port}	
Buffer file chunk files after captured image	ft buffer file 12 {buffer,file_id} 5 n hsi{hsi_folder}/compressed_cube.bip.cmpr_ chunk{chunk_amount}	Buffering remote file path hsi{hsi_folder}/compressed_cube.bip.cmpr_ chunk{chunk_amount}
	opu log	Percent: 0
	opu log	Buffering of file completed

Table 22: Test file used in testing of a more complex test, containing multiple test cases (Part 1)

Rerun case or command	End case on fail	Wait time after execution [s]	Timeout [s]
command	No	5	10
none	Yes	10	10
none	Yes	0	10
command	Yes	20	250
none	Yes	10	10
none	Yes	0	10
command	Yes	5	10
command	Yes	0	10
none	Yes	5	10
case	Yes	5	10
case	Yes	30	10
none	Yes	360	10
none	Yes	0	10

Table 23: Test file used in testing of a more complex test, containing multiple test cases (Part 2)

The data file, Code Listing 7 contains the test data for the complex test. In the data file, there are four object/value pairs that will be used. The `buffer_port` and the `file_id` are mapped together in `hypso-cli`, i.e. buffer port 22 is mapped to the file id 35, and buffer port 23 is mapped to file id 36. In the test file the command `ft clear 6`, Table 22, clears the files on on the buffer ports, while the command `ft buffer file 12 {buffer,file_id}` buffer the files from thr OPU to the specified files in the PC. The object/value pair `hsi_folder` contains the folder that that the captured Hyper Spectral Image (HSI) image, and subsequent chunks are placed into on the OPU. The object/value pair `chunk_amount` contains the amount of chunks created from the command `hsi chunkify`, and depend on the size of the HSI capture. A capture of 250 frames with software compressing, and split into 8 MB chunks, creates two chunk files which are numbered 0 and 1.

```
1
2 {
3     "buffer" : {
```

```
4           "buffer_port" : [22,23],
5           "file_id" : [35,36]
6       },
7       "chunk_amount" : [0,1],
8       "hsi_folder" : [0]
9   }
```

Code Listing 7: Data file used when testing the more complex test, containing four object/value pairs

7.4.2 Pull requests

Pull requests in both the `assembly-integration-test` and the `hypso-sw` repository, handle the same way for the complex test, as for the first simple test. The dispatch and pull request events that are used for triggering the testing are displayed in Figure 47 and Figure 48. As both events produce successful tests, the merge messages in Figure 49 and Figure 50 also look identical as the first test.

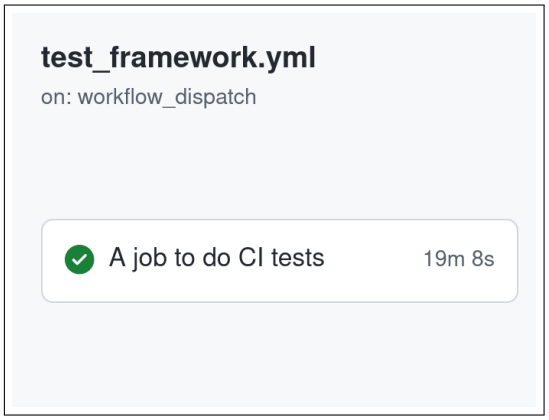


Figure 47: Result from a GitHub Actions run in the `hypso-sw` repository that has been triggered by a dispatch event. The GitHub Actions run was successful.

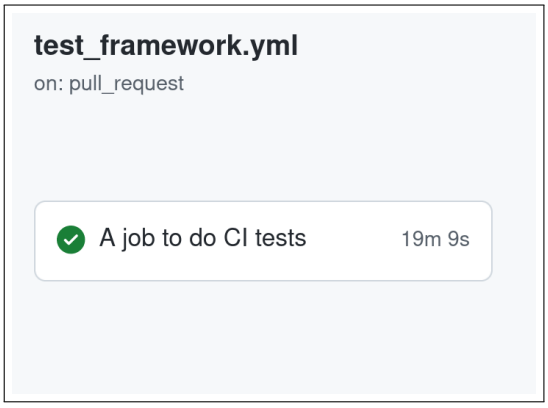


Figure 48: Result from a GitHub Actions run in the `assembly-integration-test` repository that has been triggered by a pull request event. The GitHub Actions run was successful.

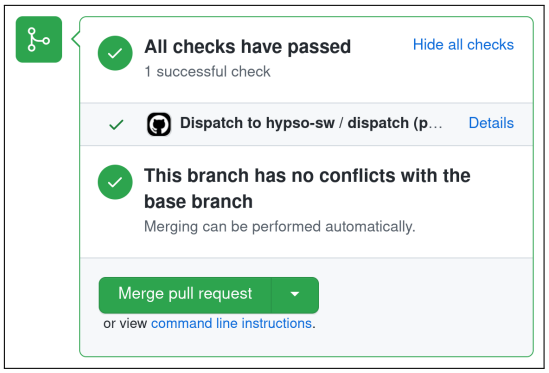


Figure 49: Merge message in `assembly-integration-test` repository when issuing a pull request. In this merge message the workflow file has executed successfully.

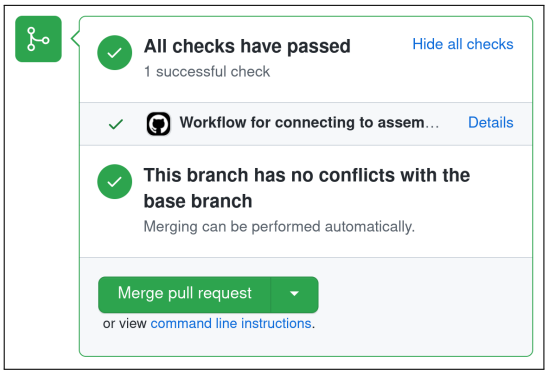


Figure 50: Merge message in `hypso-sw` repository when issuing a pull request. In this merge message the workflow file has executed successfully.

7.4.3 Result file

Executing the third, and more complex, test resulted in the result file seen in Table 24 and Table 25. In the result file, one can see that the test had a 100% pass percent, which means that every command that was executed passed. Which indicates that the framework is able to handle test that are more faithful to the actual tests the framework might be used for.

In this result file one can also see how the framework handles commands that contain more than one variable. The command `ft buffer file 12 {buffer,file_id} 5 n hsi{hsi_folder} /compressed_cube.bip.cmpr_chunck{chunk_amount}` contain three different variables, and in Code Listing 7 one can see that two of the object/value pairs for the variables contain one more value than the third. In situations like this the test script will contniue to implement copies of either command or case until every value in all object/value pairs have been used. For object/values pair that contain fewer values, their last value will be used when they run out. This is also the case when the variables in commands and expected result contain dissimilar values in their object/value pair, the last one will be used.

Date	**2022-06-11**		**Tests**	**Amount**
Git branch	HEAD		Passed	18
Git commit	3efd1e1		Failed	0
			Skipped	0
Time[UTC]	Test Results	Execution time[s]	Description	Command
12:59:08.472236	Passed	0.0009264945983886719	Clear hsi folder on the OPU	shell remote oneshot 12 5 rm -r hsi0
12:59:13.480753	Passed	0.03584694862365723	Turn of hsi camera and capture an image	shell remote oneshot 4 5 output 8 1 0
12:59:23.529297	Passed	0.04114580154418945		hsi gettemp
12:59:23.573582	Passed	130.33833050727844		hsi capture -n 250 -s
13:01:53.932438	Passed	0.0359196662902832		shell remote oneshot 4 5 output 10 1 0
13:02:03.980481	Passed	0.0326995849609375		csp ping 12
13:02:04.015861	Passed	0.033629655838012695		hsi chunkify 0 8 compressed_cube.bip.cmpr
13:02:09.056215	Passed	2.47528338432312		opu list
13:02:11.534082	Passed	2.972672939300537		opu list
13:02:14.509304	Passed	0.0009326934814453125		ft download cancel 6
13:02:19.516776	Passed	0.0009195804595947266	Request the buffer file on the PC to be cleared	ft clear 6 22
13:02:24.524881	Passed	0.001065969467163086	Request the buffer file on the PC to be cleared	ft clear 6 23
13:02:29.532391	Passed	0.03162980079650879	Buffer file chunk files after captured image	ft buffer file 12 35 5 n hsi0/compressed_cube.bip.cmpr_chunk0
13:02:59.596634	Passed	2.124535083770752		opu log
13:09:01.813439	Passed	1.432903528213501		opu log
13:09:03.249142	Passed	0.03159165382385254	Buffer file chunk files after captured image	ft buffer file 12 36 5 n hsi0/compressed_cube.bip.cmpr_chunk1
13:09:33.312591	Passed	2.1749751567840576		opu log
13:15:35.589216	Passed	1.446702241897583		opu log

Table 24: Result file from complex test, where all command passed. (Part 1).

Percentage				
100.0%				
0.0%				
0.0%				
Expected Result	Rerun case or command	End case on fail	Wait time after execution [s]	Timeout [s]
	command	No	5	10
OK	none	Yes	10	10
Temperature is	none	Yes	0	10
created folder: hsi0	command	Yes	20	250
OK	none	Yes	10	10
Ping received from OPU	none	Yes	0	10
Found file, starting partitioning	command	Yes	5	10
hsi0/compressed_cube.bip.cmpr_chunk0	command	Yes	0	10
hsi0/compressed_cube.bip.cmpr_chunk1	command	Yes	0	10
	none	Yes	5	10
	case	Yes	5	10
	case	Yes	5	10
Buffering remote file path hsi0/compressed_cube.bip.cmpr_chunk0	case	Yes	30	10
Percent: 0	none	Yes	360	10
Buffering of file completed	none	Yes	0	10
Buffering remote file path hsi0/compressed_cube.bip.cmpr_chunk1	case	Yes	30	10
Percent: 0	none	Yes	360	10
Buffering of file completed	none	Yes	0	10

Table 25: Result file from complex test, where all command passed. (Part 2).

8 Discussion

This section discusses the analysis done, and the implemented test framework, whether it fulfilled the requirements set in Section 5, or not.

8.1 Analysis

The work done in this thesis started with the analysis of the current HYPSO test setup, and then the creation of the requirements that the new framework should adhere to, see Section 5. An analysis of existing test automation frameworks were performed. What quickly became apparent for the author, was that some of the frameworks mentioned in Section 3.3, would not work for `hypso-cli` with the requirements. For example, tests created with the Linear Testing Framework would cause too much maintenance to be efficiently used by the HYPSO team, as well as not being able to utilize multiple sets of test data. The Modular Based Testing Framework would not fit well for HYPSO either, as `hypso-cli` is not split into modules, but instead functions as one large cohesive application, and the main requirement, **REQ-0** in Table 7, for the new framework was that it should imitate normal usage of `hypso-cli`.

Most of the commands in `hypso-cli` have dissimilar functionalities, and need specific input data to function properly, which means that Library Architecture Testing Framework would not be suitable as a framework either. This left the Data-Driven Testing Framework, and the Keyword-Driven Testing Framework as the two only options left, and since the Keyword-Driven Testing Framework is an improved version of the Data-Driven Testing Framework, it was chosen. The downside with the Keyword-Driven Testing Framework is its scalability, and its complexity. This framework was therefore modified by the author to mitigate these issues, and make it fit better with the usage of `hypso-cli`. The new framework is still quite complex, as it contains a lot of functionality, but the scalability is improved, as maintaining and adding new keywords is not required when new features are implemented to `hypso-cli`.

From the analysis, it also became evident that the new framework had to utilize GitHub Actions instead of Jenkins, as Jenkins has proven to be unsatisfactory solution for the HYPSO team due to the threshold of knowledge needed to be efficiently able to create new tests, and maintain the framework, see Section 5.1. Additionally, since GitHub Actions is integrated with GitHub and can effectively be triggered by a number of different GitHub events, the choice on which one to choose was quite clear.

8.2 Fulfillment of Requirements

The requirements from Section 5 are reviewed to determine whether the implemented Automated Test Framework fulfills them, based on the results from the testing in Section 7. A summary of the requirements and the level of fulfillment is provided in Table 26

Req. Number	Description	Level of fulfillment
REQ-0	The test system shall imitate normal usage of hypso-cli.	High
REQ-1	Creating new tests in the test system shall be quick and effortless.	High
REQ-2	The test system shall log the results from the tests in a results file.	Medium
REQ-3	The test system shall be usable with future features on hypso-cli.	High
REQ-4	The test system shall have low maintenance.	Medium
REQ-5	The test system shall be able to test multiple sets of data with the same command.	High
REQ-6	The test system shall be able to automatically execute tests through the use of GitHub Actions.	High

Table 26: Summary of the design requirements, and the level to which they were fulfilled (ranging from Low to Medium to High).

The requirement of being able to imitate normal usage of `hypso-cli` is fully implemented. The test framework is able to input both simple and more complex commands, with specific timings, which allows the user of the framework full control of when commands are executed and what the expected output from the commands are. As the test framework interacts with `hypso-cli` the same way as a normal user would, by connecting to `hypso-cli` and then inputting commands, the framework is not affected by the addition of new features to the application, as long as the new features do not hinder the previous features in any way. Based on this, the test framework is able to fulfill **REQ-0** and **REQ-3** with a high level of fulfillment.

From the tests executed in the results, see Section 7, the test framework operates as intended. New tests are quickly made by creating a test file containing a table with commands, expected output from the commands, and specific timings for when the commands are to be executed. Creating a data file in addition, gives the possibility of adding variables to the test file, which allows testing over multiple sets of data. Creating these tests do not require much more knowledge than what is required for using `hypso-cli`. Therefore, **REQ-1**, and **REQ-5** are set to high level of fulfillment in Table 26.

After a test is executed a result file is created. This result file contains the same information as the test file, but after it has been combined with the data file. Additionally, the result file also contains information on what the result from each command were, either passed, failed, or skipped, and how many passes, fails, and skips there were in total. The test framework also enables testing of edge cases. If one wants to test a command containing a variable, where the object/value pair contains a range of values, where some should fail, while others should pass, the entire test case can be marked as pass. This is accomplished by having the entire expected result for the command can be a variable, where the object/value pair for the variable contains the expected output from `hypso-cli`. Then, when the command should fail, the expected result can represent this by containing the output from `hypso-cli` that is given when the command fails, and opposite when the command should pass. This way, the test can have a 100% pass rate, when intended commands fail, while others pass.

For every command executed on `hypso-cli` the result file contains the UTC time of when the

command was executed, and how long time it took from execution until it received the expected value. This could be used for performing non-functional testing, like performance testing, see Section 3.2.5, on the on-board software. The HYPSO satellites have limited amount of time above the ground station at each pass around the Earth. Having efficient software is therefore advantageous. Additionally, by analyzing the time execution each command multiple time, one can also check the reliability of software, see Section 3.2.6, that the software is able to continuously perform specific functions without failure over time. Based on this, the framework is able to fulfill **REQ-2** with a high level of fulfillment. While the framework does produce a result file when executed automatically, this result file is not saved over multiple executions. When utilizing the self-hosted GitHub Actions runner, the repositories are cloned to the working directory of the runner before the tests are executed. These cloned repositories are overwritten each time the runner is activated. Which means that the result file for each subsequent run of the entire test framework, will overwrite the results from the last time. An implemented storage for these should have been implemented.

The tests in Section 7 are completely integrated with GitHub Actions, and are executed automatically on pull requests to the master branch in both `hypso-sw` repository and `assembly-integration-test` repository. By having to use the menu system when adding new tests, instead of altering the workflow files on GitHub, see Section 6.5, new users do not have to learn how the syntax for the GitHub Actions workflow files work, they only need to know how to utilize the test menu, and where to look on GitHub to find the result from the execution. Additionally, by utilizing the test menu, it allows the HYPSO team the possibility of adding tests that have not been created with the test script, to the automated execution with GitHub Actions. One drawback with the current implementation of the framework, is that the merge message in the pull request in `assembly-integration-test` repository, will be marked as passed even if the workflow in the `hypso-sw` repository fails. Due to this the user is required to check the `Actions` tab in `hypso-sw` when making a pull request, which is not ideal. Nevertheless, the framework has completed the goal of the requirement **REQ-6**, and has thereby achieved a high level of fulfillment.

The requirement **REQ-4**, The test system should have low maintenance, have a medium fulfillment level. If the functionality of a command/feature is changed, all subsequent tests that utilizes the command have to be altered. Which, with a lot of tests, can take some time. Compared to the unmodified Keyword-Driven Testing Framework, where functionality of a feature is changed, then only the keyword has to be altered, and not the test files. But this is a trade-off between lowering the learning curve for utilizing the framework, and having as low maintainability as possible.

9 Conclusion

When the author joined the HYPSO team a lot of the codebase for the on-board software for the satellites had already been created, and a system for CI/CD has been set up through the use of Jenkins. This system had proved to be less than desirable due to the amount of knowledge and time needed to learn to properly utilize the system. As a lot of the members of the HYPSO team are made up of Bachelors and Master students, working on specific projects for only a few semesters, there is not a lot of time to get familiarized with both HYPSO and the test framework to the point of being able to create and execute tests.

The work in this Master's thesis documents an analysis, design, and implementation of a new automated test framework, for the purpose of operationalizing the testing setup used by the HYPSO team at the NTNU SmallSat Lab. Based on the fulfillment level of the new framework, see Section 8, the new framework and implemented framework functions as intended and improves upon the old test framework.

The new framework utilizes a Keyword-Driven Testing Framework that has been modified to fit the needs of HYPSO. With this framework, one is able to create new tests, by creating a table containing the commands that one wants to test on the `hypso-cli` application, which is used for communication with the CubeSats. The new automated test framework also contains the possibility of testing across multiple sets of data, something that the old one could not. This

makes it so that creating tests in the new framework does not require any scripting knowledge, which lessens threshold for creating tests.

Additionally, as the codebase for the HYPSO satellites reside on GitHub, the new framework has been integrated with GitHub Actions. GitHub Actions is used as a CI/CD system in place of Jenkins, to execute testing on target hardware every time the GitHub master branch is updated with new code.

10 Future Work

As a result of the design and implementation done by the author, the automated test framework is fully implemented. What remains to be done are the addition of a few features that could make the framework even better, and add more customizability to the created tests. One of these features is the addition of using the output from `hypso-cli` as a variable for following commands. In the complex test in Section 7, the test required that the folder for the files from the HSI capture, was removed from the OPU before the capture was executed. This is due to the fact that the `hsi capture` command will increment the highest numbered folder when it created a folder for the captured image, i.e. if the OPU contains folders from hsi0 up to hsi8, then the folder created with the `hsi capture` command will be hsi9. Without knowing beforehand how many folders there are on the OPU, the test script does not know in what folders the image files were placed. If the test framework could use the output from `hypso-cli` as variables, then the deletion of the folder beforehand is not required, which could shorten the amount of commands needed in the tests.

The workflow file in the `assembly-integration-test` repository, should be altered so that the merge message in a pull request exhibits the actual result from the execution of the GitHub Actions run, and not just the dispatch event sent to `hypso-sw`. This is not natively possible with the GitHub REST API, as the dispatch event is sent to the entire `hypso-sw` repository and not to specific workflows. But a solution to the problem can be achieved in a few ways. The dispatch event can send with it an input ID that will be displayed in the GitHub Actions tab in `hypso-sw`. Then, a script can be created that continuously checks for this ID and the result from the execution, and based on this the workflow in `assembly-integration-test` can either be marked as passed or failed. Another way this problem could be fixed, is by making altering the workflow in `assembly-integration-test` to make it function similarly as to `hypso-sw`, where it will copy repositories, and execute similarly. The GitHub Actions runs, will then not be centralized, and one has to make sure both repositories utilize the same GitHub runner to prevent two runners accessing `hypso-cli` at the same time, which could cause unexpected race conditions.

Between each automatic execution of the framework, the result files should be saved in a location that persist over multiple executions of GitHub Actions. Which would allow checking the progress of the codebase over time more easily. This location could for example be a separate repository, where every time the workflow file in `hypso-sw` acquires a 100% pass rate, it could push the result file from the local directory created on the GitHub runner, up to the new repository.

In addition to the new features, the most important future work that remains is the creation of the tests files and data files that are to be executed with the framework. A study has to be done to find out which commands should be executed to give the highest amount of test coverage, while spending the least amount of time. As seen in the complex test in result, Figure 47 in Section 7, the test took just over 19 minutes to execute. When having multiple complex tests, the time can quickly add up, which might not ideal.

References

[1] Kirsten Aebersold. *Test Automation Frameworks*. https://smartbear.com/learn/automated-testing/test-automation-frameworks/. Accessed: 2022-05-13.

[2] Nano Avionics. *6U nanosatellite bus M6P*. https://nanoavionics.com/small-satellite-buses/6u-nanosatellite-bus-m6p/. Accessed: 2022-05-13.

[3] Nano Avionics. *Nano and Micro Satellite buses*. https://nanoavionics.com/buses/. Accessed: 2022-05-13.

[4] Avnet. *PicoZed*. https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/picozed/. Accessed: 2022-05-13.

[5] Bhavana Bhavar. *What are Test Automation Frameworks and Types*. https://www.clariontech.com/blog/what-are-test-automation-frameworks-and-types. Accessed: 2022-05-13.

[6] Thomas Halvard Bolle. *Payload Hardware In The Loop Testing of Satellite Operations*. 2021.

[7] Pierre Bourque and Richard E. (Dick) Fairley. *Guide to the software engineering body of knowledge*. Vol. 3.0. IEEE Computer Society, 2014.

[8] Marine Mammal Commision. *Climate Change and the Arctic*. https://www.mmc.gov/priority-topics/arctic/climate-change/. Accessed: 2022-05-28.

[9] Steve Corrigan. *Introduction to the Controller Area Network (CAN)*. Texas Instruments, 2002 - Revised 2016.

[10] Devmountain. *Git vs. GitHub: What is the difference?* https://devmountain.com/blog/git-vs-github-whats-the-difference/. Accessed: 2022-05-15.

[11] Docker. *Docker Overview*. https://docs.docker.com/get-started/overview/. Accessed: 2022-05-13.

[12] Docker. *Use containers to Build, Share and Run you applications*. https://www.docker.com/resources/what-container/. Accessed: 2022-05-13.

[13] Forcepoint. *The OSI Model Defined*. https://www.forcepoint.com/cyber-edu/osi-model. Accessed: 2022-05-15.

[14] Git. *Git*. https://git-scm.com/. Accessed: 2022-05-15.

[15] GitHub. *About branches*. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches. Accessed: 2022-05-15.

[16] GitHub. *actions/setup-python*. https://github.com/actions/setup-python. Accessed: 2022-05-13.

[17] GitHub. *Adding self-hosted runners*. https://docs.github.com/en/actions/hosting-your-own-runners/adding-self-hosted-runners. Accessed: 2022-05-13.

[18] GitHub. *Checkout V3*. https://github.com/marketplace/actions/checkout. GitHub Action. Accessed: 2022-05-15.

[19] GitHub. *Events that trigger workflows*. https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows. Accessed: 2022-05-15.

[20] GitHub. *What is GitHub Actions*. https://resources.github.com/downloads/What-is-GitHub.Actions_.Benefits-and-examples.pdf. Accessed: 2022-05-20.

[21] Amund Gjersvik. *Breakout Board V3 ICD*. 2020.

[22] Hybesis - H.urna. *Pull Request Workflow with Git — 6 steps guide*. https://medium.com/@urna.hybesis/pull-request-workflow-with-git-6-steps-guide-3858e30b5fa4. Accessed: 2022-05-15.

[23] Thomas Hamilton. *Reliability Testing Tutorial: What is, Methods, Tools, Example*. https://www.guru99.com/reliability-testing.html. Accessed: 2022-04-23.

[24] Thomas Hamilton. *Static Testing vs Dynamic Testing: What's the Difference?* https://www.guru99.com/static-dynamic-testing.html. Accessed: 2022-04-12.

[25] Thomas Hamilton. *What is BLACK Box Testing? Techniques, Example & Types*. https://www.guru99.com/black-box-testing.html. Accessed: 2022-04-30.

[26] Thomas Hamilton. *What is Functional Testing? Types & Examples (Complete Tutorial)*. https://www.guru99.com/functional-testing.html. Accessed: 2022-04-30.

[27] Thomas Hamilton. *What is Non Functional Testing? Types with Example*. https://www.guru99.com/non-functional-testing.html. Accessed: 2022-04-16.

[28] Software Testing Help. *Most Popular Test Automation Frameworks With Pros And Cons Of Each - Selenium Tutorial #20*. https://www.softwaretestinghelp.com/test-automation-frameworks-selenium-tutorial-20/. 2022-05-05.

[29] Software Testing Help. *The Differences Between Unit Testing, Integration Testing And Functional Testing*. https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/. Accessed: 2022-05-13.

[30] Nicki Holmyard. *Killers at sea: Harmful algal blooms and their impact on aquaculture*. https://www.globalseafood.org/advocate/killers-at-sea-harmful-algal-blooms-and-their-impact-on-aquaculture/. Accessed: 2022-05-28.

[31] Magne Hov. *Design and Implementation of Hardware and Software Interfaces for a Hyperspectral Payload in a small Satellite*. Master's thesis, NTNU, 2019.

[32] Aidan Hobson Sayers Ian Miell. *Docker in Practice*. Second Edition. Manning Publications Co, 2019.

[33] Ecma International. *The JSON Data Interchange Syntax*. 2nd Edition. 2017.

[34] Jenkins. *Jenkins User Documentation*. https://www.jenkins.io/doc/. Accessed: 2022-05-15.

[35] Kubos. *The CubeSat Space Protocol*. https://docs.kubos.com/1.2.0/apis/libcsp/csp_docs/overview.html. Accessed: 2022-05-15.

[36] Glenn Lee. *Types of Software Testing: Differences and Examples*. https://www.loadview-testing.com/blog/types-of-software-testing-differences-and-examples/. Accessed: 2022-05-13.

[37] Olga Zakutnyaya Lev Zelenyi. *The "Simplest Satellite" That Opened Up the Universe*. https://www.americanscientist.org/article/the-simplest-satellite-that-opened-up-the-universe. Accessed: 2022-05-28.

[38] Sarah Loff. *CubeSats Overview*. https://www.nasa.gov/mission_pages/cubesats/overview. Accessed: 2022-05-28.

[39] Tuva Okkenhaug Moxnes. *A common Software framework for a CubeSat with multiple payloads*. Master's thesis, NTNU, 2021.

[40] Dennis Langer Roger Birkeland. *HYPSO-UM-004 Manual for FlatSat and LidSat*. Internal Document. Non-published. 2022.

[41] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. SolidMatrix Technologies, Inc., 2022.

[42] Simplilearn. *An Introduction to Subprocess in Python With Examples*. https://www.simplilearn.com/tutorials/python-tutorial/subprocess-in-python. Accessed: 2022-05-13.

[43] GitHub Script team. *actions/github-script*. https://github.com/marketplace/actions/github-script. Accessed: 2022-05-15.

[44] HYPSO team. *CLAW-1 Assembly, Integration and Test Plan*. 2020.

[45] HYPSO team. *HYPSO SW Design Report*. 2020.

[46] HYPSO-SW team. *assembly-integration-test*. https://github.com/NTNU-SmallSat-Lab/assembly-integration-test. GitHub repository. Accessed: 2022-05-13.

[47] HYPSO-SW team. *Hardware-in-the-loop testing*. https://github.com/NTNU-SmallSat-Lab/hardware_in_loop. GitHub repository. Accessed: 2022-05-13.

[48] HYPSO-SW team. *hypso-sw*. http://github.com/NTNU-SmallSat-Lab/hypso-sw. GitHub repository. Accessed: 2022-05-13.

[49] HYPSO-SW team. *hypso-sw-build-check*. https://github.com/NTNU-SmallSat-Lab/hypso-sw-build-check. GitHub repository. Accessed: 2022-05-13.

[50] NTNU SmallSat Lab team. *Hyperspectral imager*. https://www.ntnu.edu/web/smallsat/mission-hyper-spectral-camera. Accessed: 2022-05-13.

[51] NTNU SmallSat Lab team. *Software Defined Radio*. https://www.ntnu.edu/web/smallsat/mission-software-defined-radio. Accessed: 2022-05-13.

[52] Testim. *Your Complete Guide to Test Automation Frameworks*. https://www.testim.io/blog/test-automation-frameworks/. Accessed: 2021-07-23.

Appendix

A Hypso-cli commands

Command	Description
clear	Clear the terminal
csp ...	CSP-specific commands.
csp buffers	request free buffers from CSP node.
csp conn	Print CSP connection table for this node.
csp debug	Toggle CSP output debug levels.
csp hello	Send a ‘hello world‘ CSP packet.
csp if	Print CSP interfaces for this node.
csp init ...	Commands for initialising CSP.
csp mem	request free memory from CSP node.
csp ping	Ping a CSP node.
csp reboot	Request a CSP node to reboot.
csp route	Print CSP routing table for this node.
csp shutdown	Request a CSP node to shutdown.
csp uptime	request uptime from CSP node.
eps ...	EPS specific commands.
eps tm	Request and print EPS general telemetry.
eps wdreset	Reset the counter for the ground system watchdog.
exit	Exit this CLI.
ft ...	File Transfer specific commands.
ft buffer file	Request a file to be buffered to the PC.
ft check	Check the integrity or presence of file entries.
ft clear	Request a file to be cleared.
ft deregister	Deregister link for a file ID.
ft extract	Request the extraction of a formatted file.
ft download cancel	Send request to cancel ongoing download.
ft extract	Request the extraction of a formatted file.
ft format	Request formatting of a file.
ft info	Request metadata for a file.
ft list	Request file listing from a node.
ft prepare	Create new formatted file from existing file.
ft register	Register a link for a file path to a file ID.
ft upload file	Upload a formatted file.
help	Print helptext for a command.
shell	Run a local shell command. Enter ’help shell’ for subcommands
shell remote	Enter remote shell mode for a specified node.

Table 27: Commands in hypso-cli application, with description

Command	Description
hsi ...	CLAW-1 specific commands.
hsi capture	Initiates a cube capture sequence.
hsi chunkify	Segments image cube for transfer to payload controller
hsi compress	Perform compression on a binned cube file.
hsi dmatest	Tests the CubeDMA module.
hsi gettemp	Poll the HSI camera temperature.
hsi metazip	Request meta data files from hsi capture to be compressed.
list	List commands, or sub-commands of a specific command.
ls	ls -l –color=always
opu ...	Commands for controlling the OPU.
opu caminfo	Get information about all ueye cameras that are detected by the OPU.
opu check	Compare local and remote checksums.
opu download	Download a file from the OPU.
opu exit	Request the opu-services process to exit.
opu git	Get branch and commit of opu-services.
opu lastcmd	Request the last command received by one of the opu services.
opu list	List files in OPU's current directory.
opu log	Get boot-log from opu.
opu restart	Requests to restart into a specific opu-services
opu setid	Change the camera ID of a connected ueye camera. Valid IDs range from 1 to 254.
opu setip	Change the IP configuration of a connected ueye camera identified by device ID.
opu settime	Request and print unix time from M6P node, and set system time.
opu shutdown	Shuts down OPU
opu status	Get status of OPU (simple telemetry).
opu telemetry	Get current telemetry status from opu-services.
opu tmlog	Turn on/off the telemetry logging on the OPU.
opu update	Update a file on the OPU.
opu upload	Upload a file to the OPU.
pl ...	Commands for controlling the OPU.
pl check	Compare local with remote checksum on specified PL .
pl download	Download a file from the specified payload.
pl exit	Request the services process of the specified payload to exit.
pl git	Get git branch and commit of specified payload.
pl lastcmd	Request the last command received by one of the services on the specified payload.
pl list	List files in a payload's current directory.
shell	Run a local shell command. Enter 'help shell' for subcommands
shell remote	Enter remote shell mode for a specified node.

Table 27: Commands in hypso-cli application, with description (Continued)

Command	Description
pl restart	Requests the specified payload to restart into a specific pl-services
pl settime	Request and print unix time from M6P node of given address. Default address is 4, EPS. Set PL system time
pl shutdown	Shuts down specified PL
pl status	Get status of specified payload (simple telemetry).
pl tmlog	Turn on/off the telemetry logging on the specified PL.
pl upload	Upload a file to the specified PL.
q	Exit this CLI.
rgb ...	RGB specific commands.
rgb capture	Makes the OPU perform one RGB image capture
rgb configfile	Load a different camera configuration file
rgb configure	Set new rgb parameter values.
rgb deinit	Deinitialize the camera.
rgb init	Initialise RGB camera.
rgb print	Print current parameter configuration to OPU's stdout
sdr ...	Commands for controlling the SDR.
sdr check	Compare local and remote checksums.
sdr download	Download a file from the SDR.
sdr exit	Request the sdr-services process to exit.
sdr git	Get branch and commit of sdr-services.
sdr lastcmd	Request the last command received by one of the sdr services.
sdr list	List files in SDR's current directory.
sdr log	Get boot-log from sdr.
sdr restart	Requests to restart into the sdr-services specified
sdr settime	Request and print unix time from M6P node of given address. Default address is 4, EPS. Set SDR system time
sdr shutdown	Shuts down SDR
sdr status	Get status of SDR (simple telemetry).
sdr telemetry	Get current telemetry status from sdr-services.
sdr tmlog	Turn on/off the telemetry logging on the SDR.
sdr update	Update a file on the SDR.
sdr upload	Upload a file to the SDR.
sdr xadc	Get current xadc status from sdr-services.
shell	Run a local shell command. Enter 'help shell' for subcommands
shell remote	Enter remote shell mode for a specified node.

Table 27: Commands in hypso-cli application, with description (Continued)

B Hypso-sw-build-check

B.1 Dockerfile

```
# Use Ubuntu 20.04 as a base image
FROM ubuntu:20.04

# Fix timezone issue
ENV TZ=Europe/Oslo
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc
    /timezone


# Install general tools and libraries
RUN dpkg --add-architecture i386 && apt-get update && apt-get install
     -y \
    build-essential \
    apt-utils \
    sudo \
    locales \
    git \
    clang-tools \
    cmake

#Make a HYPSO user
RUN adduser --disabled-password --gecos '' hypso && \
    usermod -aG sudo hypso && \
    echo "hypso ALL=(ALL) NOPASSWD: ALL" >> /etc/sudoers

    RUN locale-gen en_US.UTF-8 && update-locale

# Install necessary libraries to download, compile and install nng
RUN apt-get update && apt-get install -y \
    ninja-build \
    python

# Compile and install nng. Remove source files afterwards
RUN git clone https://github.com/nanomsg/nng /home/hypso/nng && \
    cd /home/hypso/nng && \
    git checkout v1.5.2 && \
    mkdir build && \
    cd build && \
    cmake -G Ninja .. && \
    ninja && \
    ninja install && \
    rm -rf /home/hypso/nng

# Download dependencies of building libsocketcan
Run apt-get install -y \
    autotools-dev \
    autoconf \
    libtool

# Download packages required to build hypso-sw
RUN apt-get install -y \
    check \
    make

```

```
53
54    # Download packages for cross compiling arm
55    RUN apt-get install -y \
56        binutils-arm-linux-gnueabihf \
57        gcc-arm-linux-gnueabihf
58
59    # Download packages for cross compiling arm 64-bit (aarch64)
60    RUN apt-get install -y \
61        gcc-aarch64-linux-gnu \
62        binutils-aarch64-linux-gnu
63
64    # Packages for producing doxygen output. F.ex callgraphs
65    RUN apt-get update && apt-get install -y \
66        doxygen \
67        graphviz
68    USER hypso
69    WORKDIR /home/hypso/
70
71    COPY entrypoint.sh /entrypoint.sh
72
73    ENTRYPOINT ["/entrypoint.sh"]
```

Code Listing 8: Dockerfile in hypso-sw-build-check repository

B.2 Entrypoint.sh

```
1   #!/bin/bash
2
3  make clean;
4  RESULT=$?
5  if [ $RESULT -eq 0 ]; then
6    echo make clean success
7  else
8    echo make clean failed
9    exit 1
10 fi
11
12 make;
13 RESULT=$?
14 if [ $RESULT -eq 0 ]; then
15   echo make success
16 else
17   echo make failed
18   exit 2
19 fi
20
21 make ARCH=arm;
22 RESULT=$?
23 if [ $RESULT -eq 0 ]; then
24   echo make arm success
25 else
26   echo make arm failed
27   exit 3
28 fi
29
30
31 make ARCH=arm64;
```

```
32 RESULT=$?
33 if [ $RESULT -eq 0 ]; then
34   echo make arm success
35 else
36   echo make arm for 64-bit failed
37   exit 5
38 fi
39
40 make test;
41 RESULT=$?
42 if [ $RESULT -eq 0 ]; then
43   echo check test success
44 else
45   echo check failed
46   exit 4
47 fi
```

Code Listing 9: entrypoint.sh in hypso-sw-build-check repository

C PicoZed specifications

SPECIFICATION	DESCRIPTION
SOC OPTIONS	• XC7Z010/20-1CLG400 • XC7Z015/30-1SBG485
MEMORY	• 1 GB of DDR3 SDRAM • 128 Mb of QSPI • 4 GB eMMC
CONNECTIVITY	• LPC • USB2.0 OTG • Tri-Ethernet • SFP+ • SMA HDMI • PCIe • 4 GTP/GTX ports (7015/7030 model) (see Other)
EXPANSION	• 100 pin HIGH DENSITY (x3)
VIDEO DISPLAY	• HDMI (Soft IP)
USER INPUTS	• 7Z010 Version -113 User I/O (100 PL, 13 PS MIO) ◦ PL I/O configurable as up to 48 LVDS pairs or 100 single-ended I/O • 7Z015 Version-148 User I/O (135 PL, 13 PS MIO) ◦ PL I/O configurable as up to 65 LVDS pairs or 135 single-ended I/O ◦ 4 GTP transceivers • 7Z020 Version-138 User I/O (125 PL, 13 PS MIO) ◦ PL I/O configurable as up to 60 LVDS pairs or 125 single-ended I/O • 7Z030 Version-148 User I/O (135 PL, 13 PS MIO) ◦ PL I/O configurable as up to 65 LVDS pairs or 135 single-ended I/O ◦ 4 GTX transceiver
	• Xilinx PC4 JTAG (with supported hardware)
POWER	• 12VDC
CERTIFICATION	• CE
DIMENSIONS	• 2.25"x3.7"x0.366"
CONFIGURATION MEMORY	• 128 Mb QSPI Flash • uSD card
ETHERNET	• 10/100/1000 Ethernet
GPIO	• See User Inputs
PCIE	• With supported carrier • PCIe x1 Gen2
USB	• USB 2.0
COMMUNICATIONS	• USB 2.0 • USB-UART • 10/100/1000 Ethernet
USER I/O	• (See User Inputs)
SOFTWARE	• PetaLinux BSP
OTHER	• Mated with a FMCv2 PCIe Carrier Card

Figure 51: Complete PicoZed specifications. The HYPSO team utilizes a PicoZed with Xilinx XC7Z030-1SBG485 SoC (System-on-Chip).

D Test_Script.py

```
#!/usr/bin/python3

from genericpath import exists
import os
import time
import sys
import re
import json
import csv
import signal
from datetime import date, datetime, timezone
import logging


path = ""
current_dir = os.getcwd()
if os.path.exists('../assembly-integration-test'):
    new_dir = "automated_test/"
    path = os.path.join(current_dir, new_dir)
elif os.path.exists('assembly-integration-test'):
    new_dir = "assembly-integration-test/automated_test/"
    path = os.path.join(current_dir, new_dir)
elif os.path.exists('../../assembly-integration-test/'):
    new_dir = "../automated_test/"
    path = os.path.join(current_dir, new_dir)

os.chdir(path)

sys.path.append('.')
sys.path.append('../tvac_automation')

import test_settings as test_s
import test_functions as test_f


from test_settings import OPU_EPS_CHANNEL

class LogFile(object):
    def __init__(self, name=None):
        self.terminal = sys.stdout
        self.logger = logging.getLogger(name)

    def write(self, msg, level=logging.INFO):
        self.terminal.write(msg)
        if len(msg) > 2:
            new_msg = re.compile(r'\x1b[^m]*m').sub('', msg)
            self.logger.log(level, new_msg)


    def flush(self):
        for handler in self.logger.handlers:
            handler.flush()

def main():
    # 0) Save test start time
    test_start_datetime = datetime.now(timezone.utc).strftime("%y%m%dT%H%M%S")

    # 1) Check inputs

    csv_file = ""
    data_file = ""
    input_check_ret, no_log = input_check()
    if input_check_ret == 1:
        csv_file = sys.argv[1 + no_log]
        test_data_file = "No test settings file inputted"
```

```
elif input_check_ret == 2:
    csv_file = sys.argv[1 + no_log]
    test_data_file = sys.argv[2 + no_log]
    with open(test_data_file, 'r') as f:
        data_file = json.load(f)
else:
    return

if no_log != 1:
    if exists(csv_file.split('.csv')[0] + '.log'):
        os.system(f"rm {csv_file.split('.csv')[0] + '.log'}")

    logging.basicConfig(level=logging.INFO,
                        format='%(asctime)s %(name)s: %(message)s',
                        datefmt='%m-%d-%y %H:%M:%S',
                        filename=csv_file.split('.csv')[0] + '.log')
    sys.stdout = LogFile('stdout')
    sys.stderr = LogFile('stderr')



print(test_f.strGreen("Inputted test file:") + test_f.strCyan(f"{csv_file}"))
print(test_f.strGreen("Inputted settings file:") + test_f.strCyan(f"{test_data_file}"))


# 2) Go through test file and settings file
#    Extend values from test file if required

print("Reading test file")
test_info = read_csv_file_to_list(csv_file)
test_case_start_number = []

for i in range(1, len(test_info)):
    if test_info[i][0] != '':
        test_case_start_number.append(i)
test_case_start_number.append(len(test_info))


print("Extending test file depending on variables used")
test_info = extend_test_file_with_variables(test_info, test_case_start_number, data_file, no_log)

test_case_start_number = []
for i in range(1, len(test_info)):
    if test_info[i][0] != '':
        test_case_start_number.append(i)
test_case_start_number.append(len(test_info))

# 3) Starting and rebooting hypso_cli
#    And also getting git version
hypso_cli = test_f.get_hypso_cli()
git_version = starting_hypso_cli(hypso_cli)


# 4) Testing

test_results = []
for i in range(len(test_info)):
    test_results.append(['','',''] + test_info[i])
test_results[0][0] = "Time[UTC]"
test_results[0][1] = "Test Results"
test_results[0][2] = "Execution time[s]"

pass_number = 0
fail_number = 0
skip_number = 0
```

```
for i in range(len(test_case_start_number)-1):
    print('')
    print("Starting testing with purpose of:")
    print(f"{test_info[test_case_start_number[i]][0]}")
    if len(test_info) < test_case_start_number[-1]:
        pass

    for j in range(test_case_start_number[i], test_case_start_number[i+1]):

        test_results[j][0] = datetime.now(timezone.utc).strftime("%H:%M:%S.%f")
        ret, execution_time = execute_commands(j, test_info)

        if ret == 1:
            test_results[j][1] = "Passed"
            pass_number = pass_number + 1
            test_results[j][2] = execution_time
        elif ret == 2:
            test_results[j][1] = "Failed"
            fail_number = fail_number + 1
            test_results[j][2] = "NA"
        elif ret == 0:
            test_results[j][1] = "Failed"
            fail_number = fail_number + 1
            test_results[j][2] = "NA"
            for n in range(j+1, test_case_start_number[i+1]):
                test_results[n][1] = "Skipped"
                skip_number = skip_number + 1
                test_results[n][0] = "NA"
                test_results[n][2] = "NA"
            break

# 5) Saving test results in its own file

# Creating result folder
path = os.getcwd()
new_dir_name = "test_result"
path = os.path.join(path, new_dir_name)
#Create test result folder if it doesn't exist
try:
    os.mkdir(path)
except FileExistsError:
    pass

print("")
#results = [["Date", date.today()],["Git branch", git_version[0]],["Git commit", git_version[1]],[]]
tot_commands = pass_number + fail_number + skip_number
pass_percent = pass_number/tot_commands * 100
fail_percent = fail_number/tot_commands * 100
skip_percent = skip_number/tot_commands * 100
results =  [["Date", date.today(), '', 'Tests','Amount','Percentage'],
        ["Git branch", git_version[0],'','Passed',pass_number,str(pass_percent)+'%'],
        ["Git commit", git_version[1],'','Failed',fail_number,str(fail_percent)+'%'],
        [" ",'','','Skipped',skip_number,str(skip_percent)+'%'], [" "], [" "]]
test_results = results + test_results
test_result_file = path + '/' + csv_file.split(".csv")[0] + "_" + test_start_datetime + ".csv"

print("Saving results in" + test_f.strCyan(test_result_file))
with open(test_result_file, 'w', encoding='UTF8') as f:
    writer = csv.writer(f)
    for item in test_results:
        writer.writerow(item)

# 6) Turning of OPU and killing hypso_cli
killing_hypso_cli(hypso_cli)

# 7) Removing log files
```

```
    print("Deleting log files starting with 220")
    os.system("rm 220*")

    # 8) Printing complete test result
    if pass_percent == 100:
        print(test_f.strGreen("Test script Passed"))
        print("All test cases completed, no skipps, no failed")
        print(test_f.strYellow("Pass Percent: 100%"))
    else:
        print(test_f.strRed("Test script Failed"))
        print("Not all test cases completed")
        print("Completed commands:", pass_number)
        print("Failed commands:", fail_number)
        print("Skipped commands:", skip_number)
        print(test_f.strYellow("Pass Percent:" + str(pass_percent) + "%"))


def extend_test_file_with_variables(test_info, test_case_start_number, data_file, no_log):
    for i in range(len(test_info)-1, 1, -1):
        if test_info[i][3] in {'case','command'}:
            new_commands = check_for_variable_in_command(test_info[i][1], data_file, no_log)
            new_results = check_for_variable_in_command(test_info[i][2], data_file, no_log)

            # Ensure new_commands and new_results are same length, if not fill it with last value
            if len(new_commands) > len(new_results):
                if new_results == []:
                    new_results = [test_info[i][2]]
                new_results.extend([new_results[-1]] * (len(new_commands) - len(new_results)))
            elif len(new_commands) < len(new_results):
                if new_commands == []:
                    new_commands = [test_info[i][1]]
                new_commands.extend([new_commands[-1]] * (len(new_results) - len(new_commands)))

            if new_commands != [] or new_results != []:
                if (test_info[i][3] == "command"):
                    temp_description = test_info[i][0]

                    for j in range(1, len(new_commands)):
                        temp = test_info[i]
                        temp[0] = ''
                        temp[1] = new_commands[len(new_commands) - j]
                        temp[2] = new_results[len(new_results) -j]
                        test_info.insert(i, temp[:])

                    # Due to how python copys lists by refrencing them instead of making copyes, it is necesarry to
                    # update the value for the first instance, last
                    # Using the deepcopy command in the library copy, would aliviate this problem. But making a deepcopy is more
timeconsuming
                    # and memory intensive
                    test_info[i][0] = temp_description
                    if new_commands != []:
                        test_info[i][1] = new_commands[0]
                    if new_results != []:
                        test_info[i][2] = new_results[0]

                if (test_info[i][3] == "case"):

                    # Start_number and end_number are used for knowing what parts of the in the commands file should be
duplicated.
                    # And then duplicating them to the correct places
                    start_number = 0
                    end_number = 0
                    for t in range(len(test_case_start_number)):
                        if i < test_case_start_number[t]:
                            start_number = test_case_start_number[t-1]
```

```
                        end_number = test_case_start_number[t]
                        break

                temp = test_info[start_number:end_number]

                for j in range(0, len(new_commands)-1):
                    temp[i-start_number][1] = new_commands[len(new_commands)-j-1]
                    temp[i-start_number][2] = new_results[len(new_results)-j-1]
                    for t in range(len(temp)):
                        test_info.insert(end_number, temp[len(temp) -t-1][:])


                # Due to how python copys lists by refrencing them instead of making copyes, it is necesarry to
                # update the value for the first instance, last
                # Using the deepcopy command in the library copy, would aliviate this problem. But making a deepcopy is more
timeconsuming
                # and memory intensive
                if new_commands != []:
                    test_info[i][1] = new_commands[0]
                if new_results != []:
                    test_info[i][2] = new_results[0]
    return test_info

def check_if_empty(list_of_lists):
    for elem in list_of_lists:
        if elem:
            return False
    return True

def check_for_variable_in_command(test_info_line, data_file1, no_log):

    # Opening the JSON data file is needed here due to python not being very memory efficient
    # If this script is executed by another script, and the data file is not opened here
    # then an error will occur when trying to read the from file.
    test_data_file = sys.argv[2 + no_log]
    with open(test_data_file, 'r') as f:
        data_file = json.load(f)


    new_commands = []
    data_from_file = []
    var_count = test_info_line.count('{')

    if '{' in test_info_line:
        s = test_info_line
        temp = (s.split('{'))[1].split('}')[0]
        temp2 = temp.split(',')
        data_from_file = data_file[temp2[0]]
        for j in range(1, len(temp2)):
            data_from_file = data_from_file[temp2[j]]

        for j in range(len(data_from_file)):
            temp3 = s.replace(('{' + temp + '}'),str(data_from_file[j]))
            new_commands.append(temp3)

    while ('{' in new_commands[-1]):
        for i in range(len(new_commands)):
            s = new_commands[i]

            temp = (s.split('{'))[1].split('}')[0]
            temp2 = temp.split(',')
            data_from_file = data_file[temp2[0]]


            for j in range(len(data_from_file)):
                try:
```

```
                temp3 = s.replace(('{' + temp + '}'),str(data_from_file[i]))
                new_commands[i] = (temp3)
            except IndexError:
                #print("hello man")
                #new_commands[i] = new_commands[i-1]
                s = new_commands[i-1]
                temp3 = s.replace(('{' + temp + '}'),str(data_from_file[-1]))
                new_commands[i] = (temp3)

            if j > len(new_commands)-1:
                s = new_commands[i-1]
                temp3 = s.replace(('{' + temp + '}'),str(data_from_file[j]))
                new_commands.append(temp3)

    return new_commands

class TimeoutException(Exception):
    pass

def Timeout_handler(signum, frame):
    raise TimeoutException()


def execute_commands(start_row, test_info):
    ret = 0
    execution_time = 0
    command = test_info[start_row][1]
    expected_result = test_info[start_row][2]
    hypso_cli = test_f.get_hypso_cli()
    print(test_f.strYellow("Executing command:") + test_f.strLightPurple(f"{command}"))
    print(test_f.strYellow("Expected result:  ") + test_f.strLightPurple(f"{expected_result}"))
    hypso_cli.stdout.flush()
    execution_time_start = time.time()
    try:
        hypso_cli.stdin.write(
            f"{command}\n"
        )
    except BrokenPipeError:
        hypso_cli = test_f.get_hypso_cli()
        hypso_cli.stdin.write(
            f"{command}\n"
        )
    time.sleep(0)


    signal.signal(signal.SIGALRM, Timeout_handler)
    signal.alarm(int(test_info[start_row][6]))  #Read timeout amount for specific command in test_info

    stdout_lines = ''

    try:
        for line in iter(hypso_cli.stdout.readline, b''):
            stdout_lines = stdout_lines + line
            if expected_result in line:
                execution_time = time.time() - execution_time_start
                print(test_f.strCyan("Command ") + test_f.strGreen(f"Completed"))
                ret = 1
                break
    except TimeoutException:
        print(test_f.strCyan("Command ") + test_f.strRed(f"Failed"))
        print(test_f.strRed("Execution of failed command resulted in:"))

        #Parse stdout data
        stdout_data = stdout_lines.split("(hypso-cli-CAN)")[-1]

        print(test_f.strPurple(stdout_data))
```

```
            if test_info[start_row][4] in {"Yes","yes"}:     # Check if test case will quit if a command fails
                print(test_f.strPurple("Quitting test case due to failure and setting in test file"))
                ret = 0
            else:
                ret = 2

        signal.alarm(0)
        if ret == 1 or ret == 2:
            #Sleep for set amount of seconds before continuing
            time.sleep(int(test_info[start_row][5]))

        hypso_cli.terminate()
        return ret, execution_time

    def starting_hypso_cli(hypso_cli):
        print("Starting hypso_cli")
        df_dict = dict()

        # Try to connect to OPU.
        # If this is not possible, the function get_reconnection_time will give a BrokenPipeError
        # And the program wil exit.
        try:
            hypso_cli.stdin.write(
                f"shell remote oneshot 4 5 output {OPU_EPS_CHANNEL} 0 0\n")
            hypso_cli.stdin.write(
                f"shell remote oneshot 4 5 output {OPU_EPS_CHANNEL} 1 0\n")
            test_f.get_reconnection_time(hypso_cli)
            df_dict['time_csp_ping_ms'] = test_f.get_csp_ping(hypso_cli)
        except BrokenPipeError:
            print(test_f.strRed("BrokenPipeError"))
            print(test_f.strRed("Was not able to start/connect to OPU"))
            exit()

        #reboot
        print("Rebooting hypso_cli")
        test_f.opu_reboot(hypso_cli)
        df_dict['clean_reboot_sec'] = test_f.get_reconnection_time(hypso_cli)
        print(
            test_f.strCyan(f"Used {df_dict['clean_reboot_sec']} seconds to reconnect to OPU"))

        test_f.cancel_shutdown(hypso_cli)

        test_f.sync_opu_time(hypso_cli)

        df_dict['git_version_start'] = test_f.get_git_version(hypso_cli)

        df_dict.update(test_f.get_opu_status(hypso_cli, key_postfix='start'))

        return df_dict["git_version_start"]

    def killing_hypso_cli(hypso_cli):
        print("Turning off OPU")
        try:
            hypso_cli.stdin.write(
                f"shell remote oneshot 4 5 output {OPU_EPS_CHANNEL} 0 0\n"
            )
        except:
            hypso_cli=test_f.get_hypso_cli()
            hypso_cli.stdin.write(
                f"shell remote oneshot 4 5 output {OPU_EPS_CHANNEL} 0 0\n"
            )
        time.sleep(1)
        print("Killing hypso_cli")
        hypso_cli.kill()

    def read_csv_file_to_list(file):
```

```
    with open(file, newline='') as f:
        reader = csv.reader(f, delimiter=';')
        data = list(reader)
    return data


def input_check():
    ret = 0
    no_log = 0
    if len(sys.argv) < 2:
        print(test_f.strRed("Not enough arguments"))
        usage_prompt()
        return ret, no_log
    elif len(sys.argv) > 4:
        print(test_f.strRed("Too many arguments"))
        usage_prompt()
        return ret, no_log
    elif sys.argv[1] == '-n':
        print(test_f.strYellow("Logging disabeled"))
        no_log = 1

    if not exists(sys.argv[1 + no_log]):
        print(test_f.strRed("Inputted test file does not exist"))
        usage_prompt()
        return ret, no_log
    elif len(sys.argv) == 3 + no_log:
        if not exists(sys.argv[2 + no_log]):
            print(test_f.strRed("Inputted data file does not exist"))
            usage_prompt()
            return ret, no_log
        else:
            ret = 2
    else:
        ret = 1

    return ret, no_log

def usage_prompt():
    print(" Usage of automated test file:")
    print("  Script to test commands on Lidsat")
    print(" ")
    print(" Input csv test file as first argument")
    print(" and a json data file as second argument(if needed).")
    print(" If a log file is not needed -n can be inputted as the first argument")
    print(" This will then prevent the log file from begin created")

    print(" ")
    print(test_f.strYellow("Example with data file:"))
    print(" ./test_script.py test_file.csv data_file.json")
    print(" ")
    print(test_f.strYellow("Example without data file:"))
    print(" ./test_script test_file.csv")
    print(" ")
    print(test_f.strYellow("Example for not creating log file:"))
    print(" ./test_script -n test_file.csv data_file.json")


if __name__ == "__main__":
    main()
```

E Test_Menu.py

```
#!/usr/bin/env python3
#############################################################
#                                                           #
#               Author:  Thomas Bolle                       #
#                 Date:  14.12.21                           #
#                                                           #
#############################################################

import curses
import functions as func
import curses.ascii

test_info = func.read_csv_file_to_list('test_info.csv')

class cursor_class:
    def __init__(self, row, col, row2, col2):
        self.row = row
        self.col = col
        self.row2 = row2    #row2 is used when the amount of tests is larger than can fit in the window
        self.col2 = col2
cursor = cursor_class(2,0, 0, 0)


def color_pair():
    curses.init_pair(1,curses.COLOR_BLACK, curses.COLOR_CYAN)   # Black, with cyan border
    curses.init_pair(2,curses.COLOR_WHITE, curses.COLOR_GREEN)  # White with green border
    curses.init_pair(3,curses.COLOR_WHITE, curses.COLOR_RED)    # White with red border
    curses.init_pair(4,curses.COLOR_CYAN, curses.COLOR_BLACK)   # Cyan with black border
    curses.init_pair(5,curses.COLOR_GREEN, curses.COLOR_BLACK)  # Green with black border
    curses.init_pair(6,curses.COLOR_YELLOW, curses.COLOR_BLACK) # Yellow with black border



# Write values from data to screen
# as well as other necessary information on the screen
def write_to_screen(stdscr, data, starting_row, starting_col):
    stdscr.erase()
    column_size = 20
    col = starting_col
    stdscr.move(starting_row, col)

    maxRow, maxCol = stdscr.getmaxyx()

    start_row = 0
    if cursor.row2 > len(data) - cursor.row:
        cursor.row2 = len(data) - cursor.row
    elif cursor.row2 <= 0:
        cursor.row2 = 0

    if cursor.row2 > 0:
        start_row = cursor.row2

    # Write indexes from data, and make border around all of the tests
    for j in range(1, len(data[0])):
        stdscr.addstr(starting_row,col,data[0][j], curses.A_BOLD)

        col = col + column_size
        stdscr.move(starting_row, col)
        stdscr.addstr('|')
        stdscr.addstr(' ')
        col = col + 2

    for i in range(col-3):
        stdscr.addstr(starting_row+1, starting_col + i, '-')
    if len(data) < maxRow:
        for i in range(col-3):
```

```
        stdscr.addstr(starting_row+1 + len(data), starting_col + i, '-')


    # Check if amount of tests is larger than size of terminal
    test_amount = maxRow - 1
    if len(data) < maxRow - 1:
        test_amount = len(data)

    # Input data from each specific test from data
    for i in range(1, test_amount):
        col = starting_col
        row = starting_row + 1

        stdscr.move(row+i, col)
        for j in range(1, len(data[i])):
            # Check for YES or NO in data, and set corresponding color to them if they match
            if data[i + start_row][j] in ["YES", "yes", "Y", "y", "Yes"]:
                data[i + start_row][j] = "YES"
                stdscr.addstr(row+i,col, data[i+start_row][j], curses.color_pair(2))
            elif data[i + start_row][j] in ["NO", "no", "N", "n", "No"]:
                data[i + start_row][j] = "NO"
                stdscr.addstr(row+i,col, data[i + start_row][j], curses.color_pair(3))
            else:
                stdscr.addstr(row+i, col, data[i + start_row][j])


            # Add spaces between each cell in every test
            col = col + column_size
            stdscr.move(row+i, col)
            stdscr.addstr('|')
            stdscr.addstr(' ')
            col = col + 2


    # Write Screen information that is not included from data
    stdscr.addstr(0, starting_col + len(data[0]) * 20, "Navigation of menu", curses.color_pair(5) | curses.A_BOLD)
    stdscr.addstr(1, starting_col + len(data[0]) * 20, "Quit: q or Q")
    stdscr.addstr(2, starting_col + len(data[0]) * 20, "Enter: enter key")
    stdscr.addstr(4, starting_col + len(data[0]) * 20, "YES and NO are toggled values")
    stdscr.addstr(7, starting_col + len(data[0]) * 20, "Change default values", curses.color_pair(5) | curses.A_BOLD)
    stdscr.addstr(8, starting_col + len(data[0]) * 20, "Press D or d")

    stdscr.addstr(10, starting_col + len(data[0]) * 20, "Generatge new test_sequence.csv file", curses.color_pair(5) |
curses.A_BOLD)
    stdscr.addstr(11, starting_col + len(data[0]) * 20, "Press G or g")

    stdscr.addstr(13, starting_col + len(data[0]) * 20, "Generatge new auto_test_sequence.csv file", curses.color_pair(5) |
curses.A_BOLD)
    stdscr.addstr(14, starting_col + len(data[0]) * 20, "Press A or a")

    stdscr.addstr(16, starting_col + len(data[0]) * 20, "Execute tests in test_sequence.csv file", curses.color_pair(5) |
curses.A_BOLD)
    stdscr.addstr(17, starting_col + len(data[0]) * 20, "Press R or r ")

    # Create output window
    textbox_start_row = 20
    textbox_row = 20
    textbox_col = 40
    for i in range(textbox_col + 1):
        stdscr.addstr(textbox_start_row, starting_col + len(data[0]) * 20 + i, '-')
        stdscr.addstr(textbox_start_row + textbox_row, starting_col + len(data[0]) * 20 + i, '-')

    for i in range(textbox_start_row + 1, textbox_start_row + textbox_row):
        stdscr.addstr(i, starting_col + len(data[0]) * 20, '|')
        stdscr.addstr(i, starting_col + len(data[0]) * 20 + textbox_col, '|')

    stdscr.move(cursor.row, cursor.col)
```

```
# Check for key input from the keyboard, and perform
# an action related to the specific key input
def check_key_input(stdscr, data):
    ret = 0

    maxRow, maxCol = stdscr.getmaxyx()
    cursor.row, cursor.col = stdscr.getyx()
    cursor.col = 0

    stdscr.move(cursor.row, cursor.col)
    stdscr.addstr('>', curses.color_pair(1))
    char = stdscr.getch()

    stdscr.move(cursor.row, cursor.col)
    stdscr.addstr(' ')

    if char == curses.KEY_UP:
        if cursor.row > 2:
            cursor.row = cursor.row - 1
        else:
            cursor.row2 = cursor.row2 - 1
            write_to_screen(stdscr, data, 0, 2)
    elif char == curses.KEY_DOWN:
        if cursor.row < maxRow-1 and cursor.row < len(data):
            cursor.row = cursor.row + 1
        else:
            cursor.row2 = cursor.row2 + 1
            write_to_screen(stdscr, data, 0, 2)

    elif char == ord('q') or char == ord('Q'):  # when quitting, input 'q' og 'Q'
        ret = 1

    elif char == 10:    #10 == enter key in curses
        menu_choice(stdscr, data, cursor.row, cursor.col)

    # Changing default values in the test_info.csv file
    elif char == ord('d') or char == ord('D'):
        func.write_list_to_csv_file(data, 'test_info.csv')

        # The column value of 2 + len(data[0]) * 20 + 2 is set to fit insite output window created in write_to_screen function
        write_to_screen(stdscr, data, 0, 2)
        stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, "Default values changed", curses.color_pair(6) | curses.A_BOLD)


    # Generating new test_sequence.csv file
    elif char == ord('g') or char == ord('G'):
        func.generate_test_file(data, 'test_sequence.csv')

        # The column value of 2 + len(data[0]) * 20 + 2 is set to fit insite output window created in write_to_screen function
        write_to_screen(stdscr, data, 0, 2)
        stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, "Test sequence generated", curses.color_pair(6) | curses.A_BOLD)


    # Generating new auto_test_sequence.csv file
    elif char == ord('a') or char == ord('A'):
        func.generate_test_file(data, 'auto_test_sequence.csv')

        # The column value of 2 + len(data[0]) * 20 + 2 is set to fit insite output window created in write_to_screen function
        write_to_screen(stdscr, data, 0, 2)
        stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, "Auto test sequence generated", curses.color_pair(6) | curses.A_BOLD)


    # Run trough test_sequence.csv file, and show result on screen.
        # The function run_list_of_tests_in_terminal_from_file will also create a new directory that
```

```
        # contains the output to the terminal from each test
    elif char == ord('r') or char == ord('R'):

        # Check if test_sequence.csv file does exist and is not empty.
        if func.file_exist_and_not_empty('test_sequence.csv') == 0:
            list_with_tests = func.read_csv_file_to_list('test_sequence.csv')
            stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, 'Started testing', curses.color_pair(2))
            stdscr.refresh()

            ret = []
            output = []
            for i in range(len(list_with_tests)):
                write_to_screen(stdscr, data, 0, 2)
                stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, 'Executing test:', curses.color_pair(6) | curses.A_BOLD)
                stdscr.addstr(22, 2 + len(data[0]) * 20 + 2, list_with_tests[i][1], curses.color_pair(4))
                stdscr.refresh()

                list_with_tests[i][1] = './' + list_with_tests[i][1]  #add ./ in front of the test name

                res1, res2 = func.run_test_in_terminal(list_with_tests[i][1:],
                list_with_tests[i][0])  #second argument is the path to the location of the test
                ret.append(res1)
                output.append(res2)


            func.save_test_result(list_with_tests, ret, output, 'test_result') #save result and output for each test


            write_to_screen(stdscr, data, 0, 2)
            stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, 'Testing finished', curses.color_pair(6) | curses.A_BOLD)

            result = ret.count(1)
            percent = int(result) / len(ret) * 100
            stdscr.addstr(22, 2 + len(data[0]) * 20 + 2, str(result) + " of " + str(len(ret)) + " Tests Passed.")
            stdscr.addstr(23, 2 + len(data[0]) * 20 + 2, "Pass Rate: " + str(percent) + "%")
            stdscr.addstr(24, 2 + len(data[0]) * 20 + 2, "To see output from each test,")
            stdscr.addstr(25, 2 + len(data[0]) * 20 + 2, "look in ")
            stdscr.addstr("test_result", curses.color_pair(4))
            stdscr.addstr(" directory")

        elif func.file_exist_and_not_empty('test_sequence.csv') == 1:
            write_to_screen(stdscr, data, 0, 2)
            stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, '**ERROR**', curses.color_pair(3))
            stdscr.addstr(22, 2 + len(data[0]) * 20 + 2, 'test_sequence.csv file not found', curses.color_pair(3))
            stdscr.move(cursor.row, cursor.col)
            return
        elif func.file_exist_and_not_empty('test_sequence.csv') == 2:
            write_to_screen(stdscr, data, 0, 2)
            stdscr.addstr(21, 2 + len(data[0]) * 20 + 2, '**ERROR**', curses.color_pair(3))
            stdscr.addstr(22, 2 + len(data[0]) * 20 + 2, 'test_sequence.csv file is empty', curses.color_pair(3))
            stdscr.move(cursor.row, cursor.col)
            return

    stdscr.move(cursor.row, cursor.col)
    return ret

def menu_choice(stdscr, data, row, col):
    position = 0
    column_size = 20    # size of each column on the screen
    jump = column_size + 2

    while True:
        if 1: #((cursor.row + cursor.row2) <= len(data)) and (cursor.row > 1):  # Check for pointer inside test window
            maxRow, maxCol = stdscr.getmaxyx()

            newcol = col + 2
```

```
        stdscr.erase()
        write_to_screen(stdscr, data, 0, 2)

        # Add highlighting of the users position in the test window
        stdscr.addstr(cursor.row, (cursor.col + 2) + position *jump,
            data[cursor.row + cursor.row2 - 1][position + 1] + ''.ljust(jump - 3 - len(data[cursor.row + cursor.row2 -1][position+1])),
            curses.color_pair(1))


        char = stdscr.getch()   #get input

        if char == ord('q') or char == ord('Q'):
            break
        if char == curses.KEY_RIGHT:
            if position < (len(data[cursor.row + cursor.row2 - 2])-2):
                position = position + 1

        elif char == curses.KEY_LEFT:
            if position > 0:
                position = position - 1

        if char == curses.KEY_UP:
            if cursor.row > 2:
                cursor.row = cursor.row - 1
            else:
                cursor.row2 = cursor.row2 -1
        elif char == curses.KEY_DOWN:
            if cursor.row < maxRow -1 and cursor.row < len(data):
                cursor.row = cursor.row + 1
            else:
                cursor.row2 = cursor.row2 + 1
        else:
            position = position


        if char == 10:  #10 == enter key in curses
            #Toggle YES to NO, and NO to YES without having to write it in
            if position == 0:
                if data[cursor.row + cursor.row2 -1][position+1] == 'YES':
                    data[cursor.row + cursor.row2 -1][position+1] = 'NO'

                elif data[cursor.row + cursor.row2 -1][position+1] == 'NO':
                    data[cursor.row + cursor.row2 -1][position+1] = 'YES'

            else:
                stdscr.addstr(cursor.row, newcol + jump * position, ''.ljust(jump-2))
                new_word = ''
                word = data[cursor.row + cursor.row2 -1][position + 1]
                stdscr.addstr(cursor.row, newcol + jump * position, word, curses.A_STANDOUT)

                while True:
                    curses.curs_set(1)  #turn on blinking cursor

                    char = stdscr.get_wch()
                    if (isinstance(char, str) and char.isprintable()):
                        new_word += char
                    elif char == curses.KEY_BACKSPACE:
                        new_word = word[:-1]
                    elif char == '\n':  # Enter key in wch mode
                        data[cursor.row + cursor.row2 - 1][position + 1] = word
                        curses.curs_set(0)  #turn off blinking cursor
                        break
                    word = new_word
                    stdscr.addstr(cursor.row, newcol + jump * position, ''.ljust(jump-2))
                    stdscr.addstr(cursor.row, newcol + jump * position, word)
    else:
```

```
            break

    stdscr.erase()
    write_to_screen(stdscr, test_info, 0, 2)


def main(stdscr: 'curses._CursesWindow'):
    color_pair()
    curses.curs_set(0)  #turn off blinking cursor

    write_to_screen(stdscr, test_info, 0, 2)

    while True:
        ret = check_key_input(stdscr, test_info)
        if ret == 1:
            break


if __name__ == "__main__":
    try:
        curses.wrapper(main)    #initializes curses window with standard inits
    except curses.error:
        print("**ERROR**")
        print("Your terminal is too small to fit the interface.")
        print("Please expand it and try again")
```

F Auto_test_execution.py

```
#!/usr/bin/env python3

############################################################
#                                                          #
#              Author:  Thomas Bolle                       #
#                Date:  27.01.22                           #
#                                                          #
############################################################
# Execute test_sequence.csv without using test_menu.py
# Used for automation of testing.


import functions as func
import os


def main():
    path = os.path.dirname(__file__) + "/auto_test_sequence.csv"

    if func.file_exist_and_not_empty(path) == 0:
        list_with_tests = func.read_csv_file_to_list(path)
        list_with_tests_2 = func.read_csv_file_to_list(path)
        ret = []
        output = []
        for i in range(len(list_with_tests)):
            list_with_tests[i][1] = './' + list_with_tests[i][1] #add ./ in front of the test name

            res1, res2 = func.run_test_in_terminal(list_with_tests[i][1:], list_with_tests[i][0])
            ret.append(res1)
            output.append(res2)

        func.save_test_result(list_with_tests, ret, output, 'auto_test_result')

        print("Testing Finished")

        if 0 in ret:
            print("Failed Tests:")
            for i in range(len(ret)):
                if ret[i] == 0:
                    temp = ""
                    for j in range(1, len(list_with_tests_2[i])):
                        temp = temp + " " + str(list_with_tests_2[i][j])
                    print(temp)
        print(" ")
        result = ret.count(1)
        percent = int(result) / len(ret) * 100
        print(str(result) + " of " + str(len(ret)) + " Tests Passed")
        print("Pass rate: " + str(percent) + "%")


    elif func.file_exist_and_not_empty('auto_test_sequence.csv') == 1:
        print("**ERROR**")
        print("auto_test_sqeuence.csv file not found")

    elif func.file_exist_and_not_empty('auto_test_sequence.csv') == 2:
        print("**ERROR**")
        print("auto_test_sequence.csv file is empty")

if __name__ == "__main__":
    main()
```

G Functions for Test_Menu.py and Auto_test_execution.py

```
#!/usr/bin/env python3

############################################################
#                                                          #
#              Author:  Thomas Bolle                       #
#                Date:  14.12.21                           #
#                                                          #
############################################################

import shutil
import subprocess
import tempfile
import os
import csv
import json
from shutil import rmtree


# Write list to a csv file
def write_list_to_csv_file(data, file):
    with open(file, 'w', encoding='UTF8', newline='') as f:
        writer = csv.writer(f)
        for item in data:
            writer.writerow(item)



# Run test in terminal and check for the argument "Passed".
# If passed, then return 1
# If Passed is not found, the function will return 0.
# Function also returns output from the test to the terminal.
def run_test_in_terminal(test_with_parameters, location_of_test):
    ret = 1

    path = os.path.join(os.path.dirname(__file__), location_of_test)

    with tempfile.NamedTemporaryFile(mode = 'w+t', delete = False) as tempf:
        proc = subprocess.Popen(test_with_parameters, cwd = path, stdout=tempf, stderr=tempf)

        # Wait for process to finish before continuing
        proc.wait()
        tempf.seek(0)
        output = tempf.read()

        # Check if any of the strings in pass_check exist in output
        pass_check = ["Passed", "passed", "PASSED"]
        if next((x for x in pass_check if x in output), False):
            ret = 1
        else:
            ret = 0



    return ret, output

# Save result from the tests in a new directory, with a master csv file
# containing the result for each test
# and a file for each specific tests containing the output from the test.
def save_test_result(data, result, output, dirName):
    new_dir = dirName
    current_dir = os.getcwd()
    path = os.path.join(current_dir, new_dir)

    #Try to make a directory, if it fails, due to FileExistsError,
    # then delete the directory and make a new one
    try:
```

```
        os.mkdir(path)
    except FileExistsError:
        shutil.rmtree(path) #shutil.rmtree is used for deleting directories that are not empty
        os.mkdir(path)


    #stdscr.addstr(len(data) + 8, 2, str(result) + " of " + str(len(ret)) + " Tests Passed. " + str(percent) + "% pass rate")
    temp = []
    #result_passed = result.count(1)
    temp.append(["Number of Passed tests", "Number of tests", "Pass Rate [%]"])
    temp.append([result.count(1), len(result), int(result.count(1)/len(result)*100)])
    temp.append("")
    temp.append(["Result", "Test"])

    for i in range(len(result)):
        data[i][1] = data[i][1].replace("./", '').replace('.py', '')
        with open(os.path.join(path, data[i][1]), 'w', encoding='UTF8', newline='') as f:
            print(output[i], file=f)

        if result[i] == 1:
            temp.append(["Pass", data[i][1]])
        else:
            temp.append(["Fail", data[i][1]])

        #temp.append([result[i], data[i][1]])

    # Create master file with result for each test
    write_list_to_csv_file(temp, os.path.join(path, "index.csv"))


#Run a list of tests in terminal
def run_list_of_tests_in_terminal_from_file(file):
    list_with_tests = read_csv_file_to_list(file)

    ret = []
    output = []
    for i in range(len(list_with_tests)):
        list_with_tests[i][1] = './' + list_with_tests[i][1]  #add ./ in front of the test name

        res1, res2 = run_test_in_terminal(list_with_tests[i][1:], list_with_tests[i][0])  #second argument is the path to the location of
the test
        ret.append(res1)
        output.append(res2)

    save_test_result(list_with_tests, ret, output) #save result and output for each test
    return ret


# Go through a list.
# If an element in the list contains 'YES' then the list is written to a csv file.
# Function will also remove the YES string from the list
def generate_test_file(data, file):
    temp = []
    for i in range(0, len(data)):
        if 'YES' in data[i]:
            temp.append(data[i][0:1] + data[i][2:])

    write_list_to_csv_file(temp, file)


# Read csv file and put the data into a list
def read_csv_file_to_list(file):
    with open(file, newline='') as f:
        reader = csv.reader(f, delimiter=",")
        data = list(reader)
    return data
```

```
# Check if a file exists and if it is empty
def file_exist_and_not_empty(file):
    try:
        read_csv_file_to_list(file)
        if file_empty(file):
            return 2
        return 0
    except FileNotFoundError:
        return 1

def file_empty(file):
    file_contents = read_csv_file_to_list(file)
    if not file_contents:
        return 1
    return 0
```

Thomas Halvard Bolle

Master's thesis at NTNU, 2022

NTNU
Norwegian University of
Science and Technology