École polytechnique fédérale de Lausanne

Semester project fall 2025

Master of Science in Computational Science and Engineering

Design and Implementation of On-Board Autonomy for the CHESS Flight Software

Author:
Saransh Chopra

Supervisor:
Professor Sanidhya Kashyap

Contents

1 Introduction **1**

2 Background **1**
2.1 CHESS Mission 1
2.2 Flight Software for CubeSats 2
2.3 F` 4

3 Design and Implementation of CHESS Flight Software **4**
3.1 On-Board Autonomy 5
3.1.1 EventAction Component 5
3.1.2 State Machine 6
3.1.3 Fault Detection 8
3.1.4 Fault Handling and Recovery 8

4 Results **10**

5 Future Work **10**

Appendix A SAFE state sequence version 1 **11**

Appendix B EventAction class diagram **12**

Appendix C Implementation of SAFE state's system design **13**

Acronyms

ADCS Altitude Determination and Control System. 2, 5, 6, 10

CHESS Constellation of High-Energy Swiss Satellites. 1, 2, 4, 10

CI Continuous Integration. 1, 4

CONOPS Concept of Operations. 2, 3

CubeSatTOF CubeSat-type Time-Of-Flight. 2

ECSS European Cooperation for Space Standardization. 1, 10

EPS Electrical Power System. 2, 6, 8, 10

ESA European Space Agency. 1

EST EPFL Spacecraft Team. 1, 2, 4

FDIR Fault Detection, Isolation, and Recovery. 1, 2, 3, 5, 8, 10

fpp F Prime Prime. 1, 4, 5, 7

FS Flight Software. 1, 2, 3, 4, 5, 10

FSM Finite State Machine. 1, 4, 5, 6, 7, 8, 10

FYS Fly Your Satellite. 1

GNSS Global Navigation Satellite System. 2, 6

IOD In-Orbit Demonstrator. 2

LEO Low Earth Orbit. 1

LEOP Launch and Early Operations. 8

NASA National Aeronautics and Space. 1, 4

NEST Numerical Environment for Software Testing. 4, 5, 10

OBC On-Board Computer. 1, 2, 3, 5, 6, 10

SPAD Single-Photon Avalanche Diode. 2, 6

UHF Ultra High Frequency. 2, 6, 8, 10

1 Introduction

The space sector has recently witnessed a boom in CubeSats [1], a class of nano-satellite measuring 10 cm^3 (1U) or a multiple of it. The easy accessibility and cost-effectiveness of CubeSats has enabled people from academia, industry, and other sectors to build and launch satellites as secondary payloads on a bigger launch vehicle, facilitating scientific research in space. EPFL Spacecraft Team (EST)'s flagship mission, Constellation of High-Energy Swiss Satellites (CHESS), aims to launch two 3U CubeSats on two distinct orbits (circular and elliptical) around Earth to conduct spectroscopic analysis of Earth's exospheric composition.

In orbit, a CubeSat's operations are governed by its Flight Software (FS). This software is responsible for interfacing with all hardware sub-systems, managing communications with ground stations, processing telecommands and telemetry, and ensuring overall system robustness through Fault Detection, Isolation, and Recovery (FDIR). The On-Board Autonomy of FS is responsible for maintaining nominal operations in space without ground interventions, making automated decisions, and stabilizing the satellite in the occurrence of an unexpected faults. This project aims to set up the FS for CHESS, and designs and implements the On-Board Autonomy (`EventAction`) for the CHESS mission. The FS implemented as part of this project will be deployed on *Pathfinder 0*, the first fully-integrated test satellite planned to launch in Low Earth Orbit (LEO) in 2027. Concretely, this project contributes the following to EST's CHESS mission:

- Set up the initial infrastructure and design of CHESS FS.
- Implementation of `EventAction` as an `F`` [2] component.
- Finite State Machine (FSM) governing the satellite's global operating modes and managing transitions between them.
- A design for communication between `EventAction` and different FS sub-system managers via *Triggers*, created by a continuous stream of state-changing worthy information.
- A computational algorithm to process incoming triggers, make meaningful decisions, and execute appropriate responses. These include transitioning to a global `SAFE` state or its sub-states, and initiating stabilization procedures pending ground intervention.

CHESS FS is being written in `F` v4`, an open-source framework by National Aeronautics and Space (NASA)'s Jet Propulsion Laboratory for the development and deployment of space applications. Subsequently, `EventAction` has been written using `F``, with its high level design specified in the F Prime Prime (fpp) [3] modelling language and the implementation in `C++`. The final FS image is supposed to run on an in-house On-Board Computer (OBC) with a custom Linux distribution.

Adhering to the standards laid out by NASA, each functionality in CHESS FS is accompanied with unit tests and documentation, with an eventual plan of writing integrations tests (and building a testing platform to simulate the space environment). All code changes are tested via an automated Continuous Integration (CI) pipeline, which executes test suites, formatting checks, and static code analysis. Furthermore, every addition requires a formal peer review by another member of the EST before merging into the main codebase. The design and implementation process also prioritizes compliance with standards laid out by the European Cooperation for Space Standardization (ECSS) [4]. Moreover, the designs and implementations carried out in this project have been reviewed by an European Space Agency (ESA) flight software expert (through the ESA Fly Your Satellite (FYS) program) and by a committee of experts (during EST's System Review in December 2025).

This report discusses the design and implementation of the FS and `EventAction`, and lists out the future work required on the same to meet the 2027 launch deadline.

2 Background

2.1 CHESS Mission

Given the lack of real-time and accessible measurements from Earth's upper atmosphere, EST's CHESS primarily aims to conduct spectrometric analysis in LEO to study the composition of Earth's upper atmosphere. The mission will put two CubeSats, *Pathfinder 1* and *Pathfinder 2*, in a circular and elliptical orbit around the Earth to carry out this objective. These CubeSats will produce high-resolution measurements of atmospheric

composition and precise positioning and timing information. To carry out the scientific objective, the CubeSats will deploy two payloads, a CubeSat-type Time-Of-Flight (CubeSatTOF) mass spectrometer to carry out the spectrometric analysis and a Global Navigation Satellite System (GNSS) to collect positioning and timing information.

Figure 1 shows a schematic representation of the payloads and all other sub-systems of the satellites. Along with the payloads, the CubeSats will carry five other essential sub-systems:

- Altitude Determination and Control System (ADCS) consisting of sensors and actuators to move and align the satellite using attitude determination algorithms.
- Electrical Power System (EPS) consisting of solar panels and secondary batteries to manage and distribute power.
- On-Board Computer (OBC) consisting of two sets of central processing units, memory, etc, to run the FS, which would be cross-compiled for a custom Yocto [5] based Linux distribution.
- Ultra High Frequency (UHF) antenna to transmit telemetry and receive commands to and from the ground.
- X-band antenna to transmit the scientific data collected by the payloads to the ground.

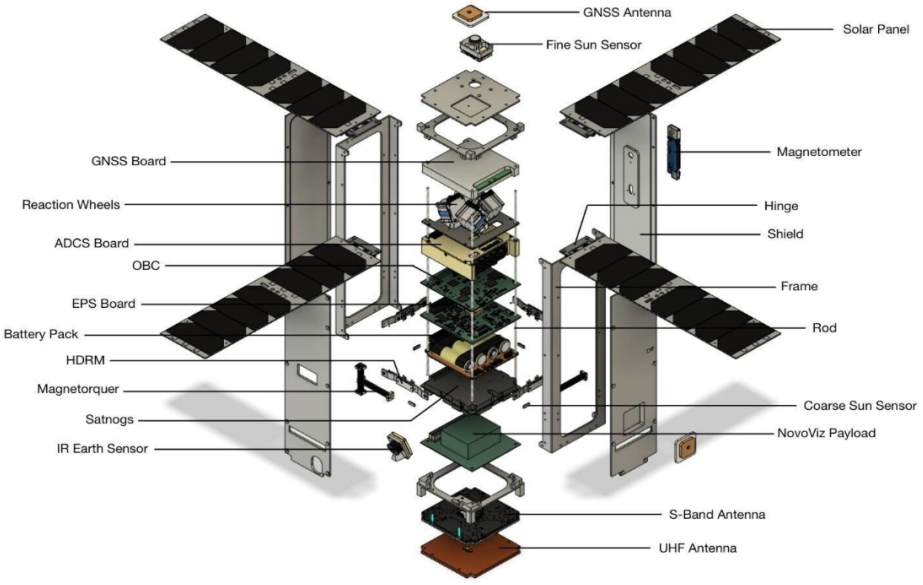


Figure 1: Structural view of Pathfinder 0 [6].

However, as a concrete test, the association has decided to include a *Pathfinder 0* satellite in the mission, which is set to be launched in early 2027. Pathfinder 0 will act the first fully-integrated test for the CHESS mission. Instead of having the initially planned CubeSatTOF, Pathfinder 0 will include a Novoviz Single-Photon Avalanche Diode (SPAD) camera to click pictures of Earth. Given that SPAD does not require a huge bandwidth to transmit the data to Earth, the satellite will include an S-band antenna instead of the X-band antenna. Finally, the Twocan board tested on EST's recent In-Orbit Demonstrator (IOD) mission [7] will serve as its OBC, providing us with a confirmation if this iteration of OBC is ready for space.

After launch, Pathfinder 0 will follow a defined Concept of Operations (CONOPS), performing its scientific objectives, maintaining nominal operations without much ground intervention, performing FDIR, and then eventually deorbiting (after 5 years of launch).

2.2 Flight Software for CubeSats

The FS of a CubeSat maintains the state of the satellite in the space, acting as a central node for communication and decision handling. The FS is broadly responsible for [8]:

- Communicating with each hardware sub-system.

- Communicating with the ground, in the form of received commands, and transmitted telemetry and events.
- Handling and storing scientific data, until it is transmitted to the ground.
- Maintaining nominal operation mode, and switching between different global satellite modes if and when required.
- Formalizing CONOPS in a machine-readable format and defining the sequence of operations for each satellite mode.
- Handling FDIR and stabilizing the satellite in the case of an unforseen fault.

To achieve these objectives, the FS is required to be modular, scalable, reliable, and re-usable [9–11]. For instance, after the lifecycle of Pathfinder 0, the FS components (especially the software abstraction over each sub-system - the sub-system managers) will be re-used for the subsequent missions. Moreover, these sub-system managers should also be easily scalable, such that if a sub-system is upgraded in the future iterations, the FS can be updated without much trouble.

Similarly, FS is required to be deterministic and reliable in every possible condition encountered in the space. The deterministic algorithm can very well lead the satellite to a `SAFE` state and wait for an intervention from the ground, but it should not create an infinite loop of state changes. F` enlists the following recommendations to respect these functional requirements, making the FS reliable and deterministic:

- No recursion; No GOTOs.
- Loops must have a fixed-bound.
- No dynamic memory allocation after initialization.
- Use `FW_ASSERT` to validate function inputs and computation.
- Restrict data to the smallest necessary scope.
- Check function return values or explicit discard with (`void`).
- Avoid the preprocessor and especially complex uses of the preprocessor.
- Prefer `Fw` and `Os` implementations. e.g. use `FW_ASSERT` and `Os::Mutex` over `cassert` and `std::mutex`.
- Compile without warnings, errors, and static analysis failures (e.g. pass continuous integration).
- Do not use `Os::Task::delay` to synchronize between threads.
- Explicit enumeration values should be specified for all values or none at all.

Furthermore, the standard embedded programming recommendations such as:

- No exceptions, code must compile with `-fno-exceptions`.
- No uses of templates nor the Standard Template Library.
- No `typeid` or run-time type information.
- Use namespaces to reduce naming conflicts.
- Use `std::numeric_limits` for mix/max values. Template implementations of `std::numeric_limits` have been approved.
- Use `static_cast`, `reinterpret_cast`, and `const_cast` instead of dangerous C-style and `dynamic_cast` casting.
- Limited use of multiple inheritance and virtual base classes is permitted.

also apply to the development of FS for space.

Architecturally, the FS for CubeSats is usually developed with a layered architecture [12], going all the way from drivers for the hardware to high level data handling and decision making (usually mission-specific). This layered architecture also allows the software to be modular and easily extendable, such that the entire software is not rewritten if a piece of underlying hardware is added, removed, or updated [13, 14]. Finally, the FS for CubeSats usually follow an architecture akin to the master-slave architecture, wherein the On-Board Autonomy makes critical decisions based on inputs from each hardware sub-system (and subsequently the sub-system managers), each sub-system manager aligns its mode of operation according to the the global satellite mode set by the On-Board Autonomy, and each sub-system only communicates exclusively with the On-Board Autonomy [15].

In case of a software failure, it should be possible to beam up a patched version of the software and reboot the OBC with the latest executable, making FS the only part of the satellite that can be patched once the satellite is in space. Thus, writing FS for space, or specifically, CubeSats, requires one to follow strict coding conventions, and choosing the right framework and programming paradigm for the FS makes it easier to not deviate from these strict requirements.

2.3 F`

F` is a C++ framework written by NASA for development and deployment of space applications. F` emphasizes *Component*-based development, isolating each flight software functionality into a stand-alone component. These components can communicate with the other components only through dedicated F` *Ports*. The framework additionally provides other interfaces for communication, safe and threaded data structures for embedded systems, a modelling language (fpp) to automatically generate a code template from systems design, pre-built widely-used components, and other utilities for testing, documentation, and ground software.

Formally, F` provides a user with the following building blocks to write software for space (or, in theory, software for any embedded system):

- **Component**: A stand-alone encapsulation of a particular software functionality. A component further contains the ports, commands, events, and telemetries it requires.
- **Port**: An interface to communicate between to components.
- **Command**: An action, usually sent by the ground, for a component to perform. Commands can also sent or scheduled by internal F` components.
- **Event**: An action performed by a component which is received by the ground in the form of logs.
- **Telemetry**: Housekeeping data sent to the ground providing the active state of a component.
- **Deployment**: A single build or executable of the FS. Each deployment consists of a topology defining the FS.
- **Topology**: A network of instances of the defined components (with thread priorities and addresses).

In addition to the high-level categorization, each of these building blocks can be further classified. For instance, the ports are further classified as *synchronous*, *asynchronous*, and *guarded* based on the type of their invocations, or *input* and *output* based on their direction of communication. Similarly, components can further be classified as *active*, *passive*, or *queued* based on their threading and queuing behavior.

Hence, F` is a robust framework with a track record of writing maintainable and modular software for space, such as the Mars Helicopter (Ingenuity) [16]. Furthermore, given that the framework was built specifically for CubeSats, small spacecrafts, and instruments, F` emerged as the perfect choice to write CHESS FS.

3 Design and Implementation of CHESS Flight Software

Following the design principles of a typical FS mentioned in 2.2, CHESS FS is designed to be modular, scalable, reliable, and re-usable. The whole software is built on a layered architecture, making it modular, scalable, and re-usable, where each driver talks to the actual hardware, the sub-system managers translate high-level commands into driver messages, and the high-level decision making and data handling algorithms only operate on the software level. Each sub-system manager has an internal FSM to manage its operations, aligning with the global operating mode set by `EventAction`'s FSM. Hence, the FS is designed to manage multiple operating modes, divided into nominal and safe modes. The global satellite operating mode transitions are handled by `EventAction`'s FSM, reacting to triggers (3.1.3), system events, and external commands. Figure 2 shows the high-level layered architecture of the CHESS FS. An instance of each of the F` components is instantiated at runtime, forming a network of components (topology) within the given F` deployment.

The FS undergoes multiple levels of testing, making it reliable and deterministic. Each version of the flight software is inspected using static analysis tools to detect potential errors. `Clang-Tidy` [17] is used to lint `CXX` and `HXX` files (with configuration adopted from the F` repository) and `Ruff` [18] to lint `Python` files. Standard pre-commit hooks [19] are also configured to lint other file formats, such as `.txt`, `.yaml`, and `.md`. Similarly, `ClangFormat` [20], `Ruff`, and other standard pre-commit hooks are used to format every file to standardize the codebase. Beyond the static analysis layer of testing, component-level testing is performed by the means of unit tests. Along every F´ component in the flight software comes an associated software design document which defines a set of requirements for the component. As many of these requirements as possible are verified by one or more unit tests, written before or together with the component to ensure maximal coverage early. These unit tests help ensure that components properly implement their requirements and behave as expected in isolation. All of these checks run automatically in a CI pipeline on GitHub on every pull request and push to the default branch (`main`). The final layer of testing being developed by the EST is Numerical Environment

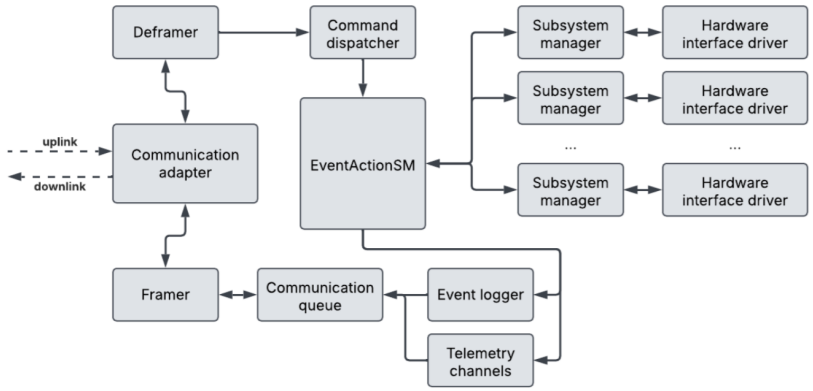


Figure 2: High level overview of the flight software architecture [6].

for Software Testing (NEST), a custom testing platform. NEST will simulate space-like conditions to perform the final integration tests for the FS.

The FS will be cross-compiled for the Yocto based Linux distribution running on OBC. The Linux distribution will provide a base software layer to deploy, configure, and update the FS whenever required. The distribution houses a bash script (daemon) that handles booting, monitoring, and updating the FS. In the case where a bug or a fault is detected in the FS, the ground operator can beam a new executable of the software to the satellite, triggering the daemon to update the FS version, making FS the only piece of the satellite which can potentially be fixed once the satellite is in space.

Finally, the FS works on system ticks produced by F`'s `BlockDriver` and `RateGroupDriver`. The FS components are invoked via F` `ActiveRateGroup` at regular time intervals. Depending on the operating mode and responses received by the drivers, the high-level components emit telemetry, events, or values capable of causing triggers (to `EventAction`). Section 3.1 describes the On-Board Autonomy (`EventAction`, the FSM, and FDIR) in detail.

3.1 On-Board Autonomy

3.1.1 EventAction Component

`EventAction` is the *brain* of FS. Written as an F` component, with the system design in fpp and the implementation in C++, `EventAction` acts as the central node for decision making and managing satellite operating modes. `EventAction` is composed of ports to communicate with other components, events and telemetries to transmit to the ground, and an FSM (3.1.2) to govern the global operating modes of the satellite.

`EventAction` has been designed to never take control of, or individually command, a sub-system, instead it only broadcasts the global operating mode of the satellite and lets the individual sub-system managers align their internal state machines with this mode. That is, if the satellite is tumbling uncontrollably, `EventAction` does not take control of ADCS. It simply passes this information (by broadcasting the global operating mode using the `sendMode` port) to every sub-system manager and waits for the tumbling rate value (3.1.3) from `ADCSManager` to go below the threshold value in time T_{max} (and then accordingly changes the global operating mode, or goes to `SAFE_COM` (3.1.4) state if the issue is not resolved in time T_{max}). Listing 1 shows an example of this broadcasting behavior, as implemented for the `GO_TO_CHARGE_MODE` command in the component. While this approach increases the number of ports and connections, it ensures explicitness in operations.

Listing 1: Mode broadcast example.

```
void EventAction::GO_TO_CHARGE_MODE_cmdHandler(FwOpcodeType opCode,
                                               U32 cmdSeq) {
    // send signal to go to charge mode
    this->eventActionSM_sendSignal_charge();
    // send event to ground
```

```
    this->log_ACTIVITY_HI_MODE_CHANGE(EventActionModes::CHARGE);
    // broadcast the mode to every component
    this->sendMode_out(0, EventActionModes::CHARGE);
    // finish with OK response
    this->cmdResponse_out(opCode, cmdSeq, Fw::CmdResponse::OK);
}
```

Furthermore, `EventAction` will usually not (except for the `SAFE_REBOOT` state) wait for the sub-system managers to do their tasks. It is expected that the tasks will be performed as required. However, the `SAFE` state (3.1.4) sequences have a fall-back if the essential tasks (such as, charging in `SAFE_CHARGE` state or detumbling in `SAFE_DETUMBLE` state) are not completed by a sub-system manager in time T_{max}, which when crossed puts the CubeSat in `SAFE_COM` state and waits for the ground to intervene.

Tables 1, 2, and 3 list every custom port, command, and event defined in `EventAction`. The ports and commands are implemented in the class itself, but the events are automatically generated by `F`` in `EventAction`'s super class `EventActionComponentBase`. Appendix B displays the complete class diagram of `EventAction`, including the members generated by `F``.

Name	Description
adcsTriggerIn	Receives triggers from ADCS sub-system manager
uhfTriggerIn	Receives triggers from UHF sub-system manager
gnssTriggerIn	Receives triggers from GNSS sub-system manager
novovizTriggerIn	Receives triggers from SPAD sub-system manager
epsTriggerIn	Receives triggers from EPS sub-system manager
obcTriggerIn	Receives triggers from OBC sub-system manager
recvFatal	Receives FATAL event announcements
sendMode	Sends current operating mode to other components

Table 1: Ports implemented in EventAction and their description.

Name	Description
GO_TO_CHARGE_MODE	Command to transition to `CHARGE` state
GO_TO_SAFE_MODE	Command to transition to `SAFE` state
GO_TO_DOWNLINK_MODE	Command to transition to `DOWNLINK` state
GO_TO_MEASURE_MODE	Command to transition to `MEASURE` state
GO_TO_COM_MODE	Command to transition to `COM` state
GO_TO_SAFE_COM_SUBMODE	Command to transition to `SAFE_COM` sub-state
GO_TO_SAFE_CHARGE_SUBMODE	Command to transition to `SAFE_CHARGE` sub-state
GO_TO_SAFE_DETUMBLE_SUBMODE	Command to transition to `SAFE_DETUMBLE` sub-state
GO_TO_SAFE_REBOOT_SUBMODE	Command to transition to `SAFE_REBOOT` sub-state

Table 2: Commands implemented in EventAction and their description.

Name	Description
MODE_CHANGE	Event for logging the new global operating mode
TRIGGER_RECV	Event for logging the trigger received

Table 3: Events implemented in EventAction and their description.

3.1.2 State Machine

`EventAction` houses an FSM to manage the satellite's global operating mode, forming a bijective relation between the FSM's states and the satellite's global operating modes. Each of these states is tailored to the

mission and has a specific purpose, with the satellite frequently jumping between these states when it enters different parts of the orbit (except for the `LEOP` and `SAFE` states). Formally, the FSM (and subsequently, the satellite) has the following states:

- `LEOP`: manages early operations, and transitions the satellite from deployment to nominal operations.
- `CHARGE`: charges the secondary batteries.
- `MEASURE`: activates the payloads and collects data.
- `COM`: enables communication with the ground segment.
- `DOWNLINK`: transmits the collected scientific data to ground.
- `SAFE`: tries to stabilize the satellite until ground intervention.
  - `SAFE_BASE`: the default `SAFE` sub-state.
  - `SAFE_COM`: prioritizes establishing communication with the ground.
  - `SAFE_CHARGE`: prioritizes charging secondary batteries.
  - `SAFE_DETUMBLE`: prioritizes detumbling the satellite.
  - `SAFE_REBOOT`: reboots the key sub-systems periodically until ground intervention.

Out of all the states, the `LEOP` and `SAFE` states are not entered as part of nominal operations. The FSM always starts in the `LEOP` state, and switches to the `CHARGE` state once all the preliminary operations (such as detumbling and deploying solar panels and UHF antenna) have been completed. The `SAFE` state can only be entered if a trigger (3.1.3) is detected by `EventAction`. The `SAFE` state sequence automatically decides which sub-state to enter, based on the trigger detected. Figure 3 shows a state diagram for the FSM, including which sub-systems are active during a particular state.

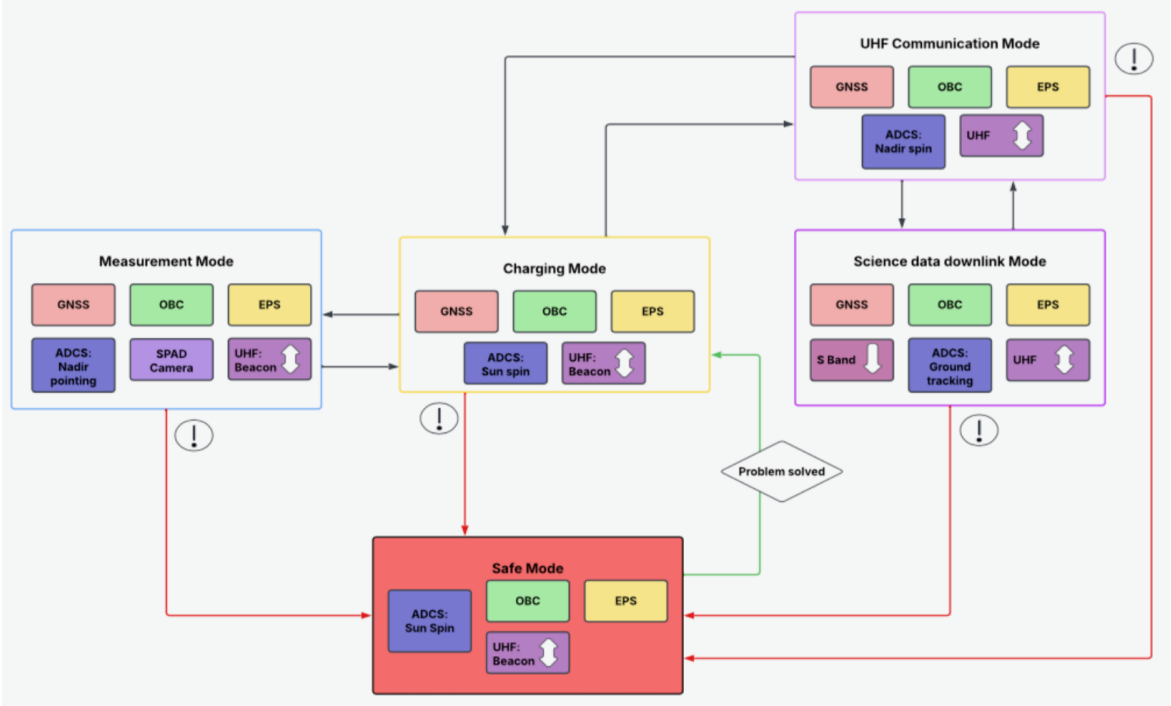


Figure 3: Global operating modes for Pathfinder 0 [6].

State transitions in the FSM are managed by *signals*, essentially C++ methods, which when called signals the FSM to switch to a particular state. Besides signals, the FSM is also equipped with *actions*, which are performed before entering or exiting a particular state. As an example, appendix C shows the system design of `SAFE` state in the FSM implemented using fpp, including the sub-states, actions, signals, and the control flow. Furthermore, appendix B displays all the class members of `EventAction`, including the FSM and the functionality related to it.

3.1.3 Fault Detection

The FDIR and decision making mechanism of `EventAction` revolves around custom triggers. Triggers are events that switch the global satellite operating mode to `SAFE` state, such as $BatteryLevel < B_{crit}$. At the beginning of this project, triggers were designed to be state-changing requests sent by a sub-system manager to `EventAction`. This micro-architecture like design allowed quick, contention-free, state changes, but theoretically gave sub-system managers the power to change states. For instance, `EPSManager` would send a request to `EventAction` to switch the global operating mode to `SAFE` if battery level drops below B_{crit}, and the `EventAction` will complete its request. However, as the project evolved, the design changed to a monolithic one, wherein, sub-system managers only send values worthy of generating triggers to `EventAction` and `EventAction` takes into consideration all such values and executes a centralized decision making algorithm.

The values worthy of generating a trigger are akin to internal *telemetry* for `EventAction`. Each sub-system manager will continuously communicate these values to `EventAction`. `EventAction` will regularly check if these values are below or above a certain threshold (generating a trigger) and act on it. Referring to the EPS example again: `EPSManager` will regularly transmit the *BatteryLevel* value to `EventAction` via the `epsTriggerIn` port. `EventAction` will regularly check this value and enter the `SAFE_CHARGE` sub-state (after processing all other trigger information in memory) if the value goes below B_{crit}. With this design, the sub-system managers are not capable of, but they play an important role in, changing the global operating mode of satellite.

Figure 4: Triggers for Pathfinder 0.

Furthermore, issues that affect only one sub-system and that can be handled by a sub-system manager internally are not escalated to `EventActions` as triggers. These issues should instead be handled internally by the sub-system manager. Figure 5 shows the final selected triggers for Pathfinder 0. The *crit* or *max* values in the diagram correspond to the thresholds *critical* enough to enter the `SAFE` state. The design also does not include the event of UHF and solar panels not opening during LEOP as triggers (as this can only occur in the `LEOP` state and will never occur again). This fault is instead handled internally in `EventAction`'s `LEOP` state. Similarly, heating up the CubeSat is not a `SAFE` state task, instead it is a hardware task automatically triggered when CubeSat's temperature goes below a threshold value. On the other hand, cooling down the CubeSat is automatically handled by the `SAFE` state sequence (all non-essential sub-systems are shut down at the beginning of `SAFE` state, automatically bringing down CubeSat's internal temperature).

3.1.4 Fault Handling and Recovery

The spacecraft enters the `SAFE` state once a trigger is detected. The rationale of Fault Handling and Recovery (and subsequently, the `SAFE` state) is to put the satellite in a stabilized mode to give as much time as possible to the ground operators to understand and fix the issue. The `SAFE` state employs a deterministic algorithm to stabilize the satellite with the help of its sub-states and priorities for each trigger. The state also prioritizes protecting valuable payloads, and the FSMs of the payload managers turns them off as soon as the spacecraft enters the `SAFE` state. Figure 6 shows the final iteration (`v4`) of the `SAFE` state sequence design, with higher priority triggers handled first. Appendix A further shows the second iteration of the design (`v1`), with `v0` omitted because of its size. With each iteration, the sequence design was made more modular and isolated; hence, the final design only includes the `SAFE` state sequence for `EventAction` with the expectation that each sub-system manager will define their own `SAFE` state sequence.

The `SAFE` state is divided into five sub-states: `SAFE_BASE` (a software abstraction over the `SAFE` state), `SAFE_CHARGE`, `SAFE_COM`, `SAFE_DETUMBLE`, and `SAFE_REBOOT`, encapsulating every `SAFE` state functionality and allowing ground operators to activate any `SAFE` state function via ground. Every `SAFE` sub-state is designed to either go back to the `SAFE` state (to process the remaining/newly detected triggers) or to the `SAFE_COM` state, which can only be exited via a command from ground; hence, `SAFE` state can only be terminated by a

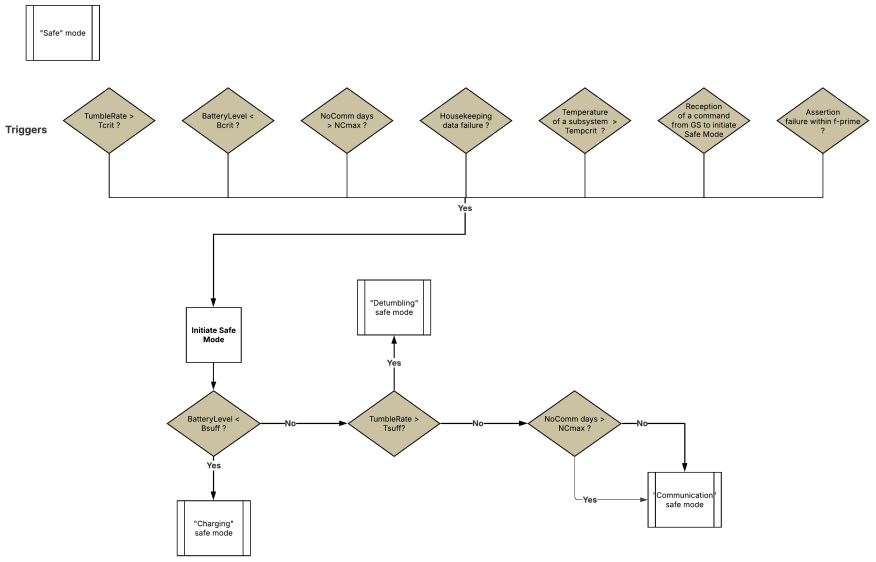
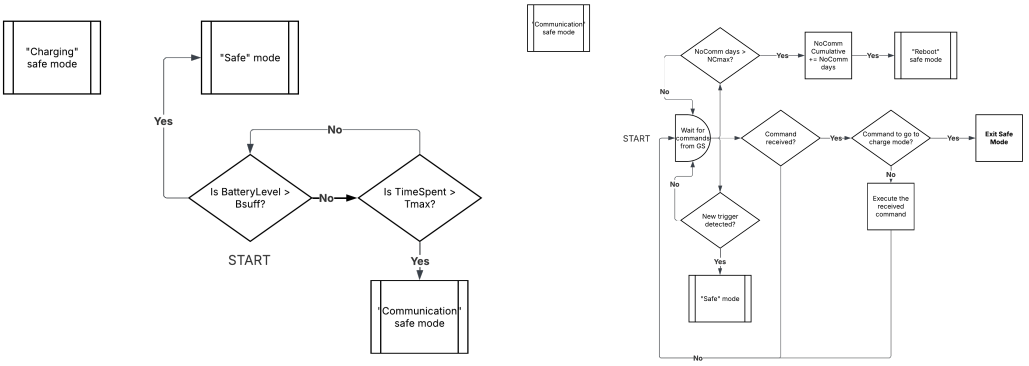
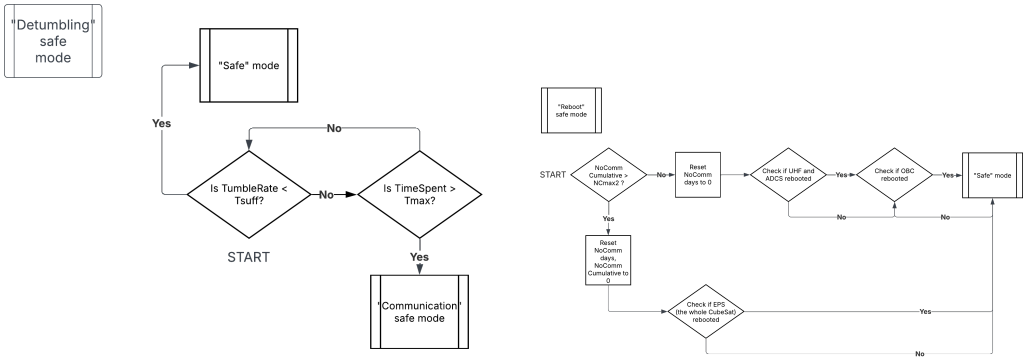


Figure 5: SAFE state (SAFE_BASE sub-state) sequence.



(a) SAFE_CHARGE sub-state sequence.

(b) SAFE_COM sub-state sequence.



(c) SAFE_DETUMBLE sub-state sequence.

(d) SAFE_REBOOT sub-state sequence.

Figure 6: SAFE sub-state sequences.

ground operator. Furthermore, `EventAction` automatically enters the `SAFE_COM` state if an essential `SAFE` state operation (such as charging in `SAFE_CHARGE` sub-state) is not performed satisfactorily in time T_{max} and waits for the ground to intervene.

The `SAFE_CHARGE` and `SAFE_DETUMBLE` states constantly try to charge the battery or detumble the satellite

until the $BatteryLevel > B_{suff}$ or $TumbleRate < T_{suff}$, respectively (where B_{suff} and T_{suff} are the threshold values *sufficient* enough to exit the sub-state). Out of the four sub-states, `SAFE_COM` and `SAFE_REBOOT` sub-states play an important role in mitigating extreme risks. Once in the `SAFE_COM` state, the FSM frequently scans for any newly detected triggers, and does not react on any unresolved triggers detected in the past, preventing an infinite loop of jumping states and making the algorithm deterministic. The `SAFE_COM` state also monitors for commands from the ground, and switches to the `CHARGE` state only when the ground commands it to do so. The `SAFE_REBOOT` state can only be entered via the `SAFE_COM` state, and the switch between the `SAFE_COM` and `SAFE_REBOOT` states is designed to be repetitive. In the `SAFE_REBOOT` state, ADCS, UHF, and OBC are rebooted iff there is no communication with ground (*NoComm* days) for $> NC_{max}$ days. Given that the switch to `SAFE_REBOOT` is repetitive, these reboots are repetitive, that is, these sub-systems are rebooted after every NC_{max} days until communication with ground is established. As the last resort, the `SAFE_REBOOT` state reboots `EPS`, and subsequently, the whole satellite if no communication is established with the ground (*NoCommCumulative* days) after NC_{max_2} days (a multiple of NC_{max}). Again, given the repetitive nature of the activation of `SAFE_REBOOT` state, the EPS reboot is also periodic; hence, the whole satellite is rebooted every NC_{max_2} days until communication with the ground is established. Following the ECSS standards, UHF (and communication with ground) is never explicitly disabled, except for when EPS is rebooted via the `SAFE_REBOOT`, rebooting the entire satellite including the UHF antenna.

Therefore, the `SAFE` state is designed from ground-up to be deterministic and reliable, positioning the satellite in a stable mode and giving the ground operators time to deploy a fix. The state strictly adheres to the ECSS standards and is also easily extendable to incorporate more stabilizing functionalities. The state can last indefinitely, provided that EPS is not damaged. The ground segment will instruct the CubeSat to exit the `SAFE` state only once the triggers are resolved and the CubeSat can safely switch back to nominal operations.

4 Results

This project successfully established the foundational infrastructure and implemented key components of the CHESS FS. Formally, the project set up the initial infrastructure and overall design of the CHESS FS, adhering to widely-adopted space standards and architectural principles. Further, the project designed and implemented the *brain* of the FS, `EventAction`, the core decision-making component. `EventAction` internally implements an FSM to govern the satellite's global operational modes, managing transitions between nominal states and various `SAFE` sub-states. Along with the state machine, FDIR has also been formalized in this project, consisting of a continuous stream of state-changing information (causing triggers), and a deterministic and reliable computational algorithm for the `SAFE` state. In summary, this project delivers the initial design and implementation of the CHESS FS and the On-Board Autonomy, providing the critical foundation necessary for maintaining nominal operations in space with minimal ground intervention.

5 Future Work

Even though the design of `EventAction` is complete, the implementation is only halfway there, as the nominal mode sequences are dependent on each hardware sub-system of CHESS, which are not completely ready as of now. Similarly, the design for FDIR is complete, but the implementation is not yet complete, and is dependent on the individual hardware sub-systems activated during the process. Particularly, the trigger causing values, timeout values, and thresholds to recover from the `SAFE` state needs to be defined by each sub-system. Further, given that ECSS standards require the spacecraft telemetry and commands to be readable by each component, it should not be required to centralize command rerouting and trigger calculation in `EventAction`. Instead, the commands can directly go to a specific component and a wrapper component (consisting of `TlmChan`, F``'s internal component to pass telemetry between components, and other custom code) can be developed to process trigger causing values and raise a trigger to the `EventAction`, which will then only deal with fault handling and recovery, and not the detection. Utmost care was taken to align this work with the ECSS standards, but comprehensive integration testing (through NEST) is required to find any loopholes, especially in the implemented decision algorithm. Finally, the initial design and implementation of CHESS FS and `EventAction` was accelerated this semester through this project, but the design and implementation of each sub-system manager and the corresponding hardware drivers is still incomplete and is required for the 2027 launch deadline.

Appendix A SAFE state sequence version 1

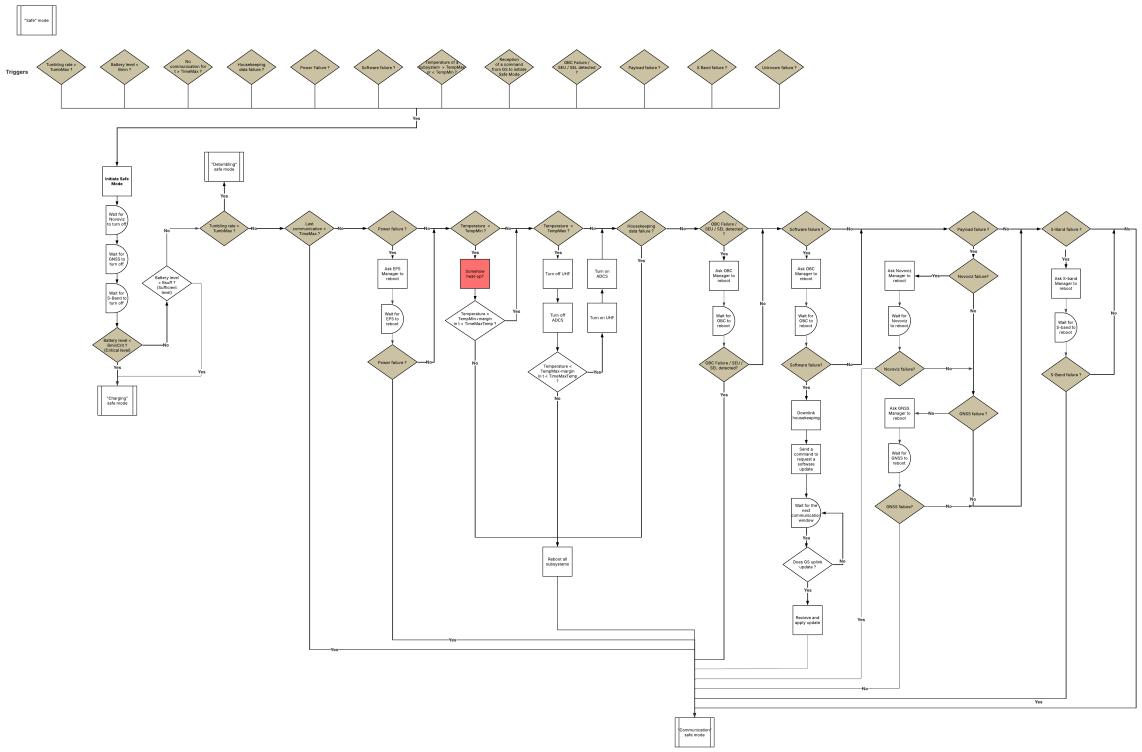


Figure 7: SAFE state sequence v1.

Appendix B EventAction class diagram

Figure 8: EventAction class diagram.

Appendix C Implementation of SAFE state's system design

Listing 2: Implementation of SAFE state's system design in fpp.

```
signal safe
action enterSafe
action exitSafe

signal safeBase
action enterSafeBase
action exitSafeBase

signal safeDetumble
action enterSafeDetumble
action exitSafeDetumble

signal safeCharge
action enterSafeCharge
action exitSafeCharge

signal safeCom
action enterSafeCom
action exitSafeCom

signal safeReboot
action enterSafeReboot
action exitSafeReboot

state SAFE {
    initial enter SAFE_BASE

    state SAFE_BASE {
        on safeCharge do { enterSafeCharge } enter SAFE.SAFE_CHARGE
        on safeCom do { enterSafeCom } enter SAFE.SAFE_COM
        on safeDetumble do { enterSafeDetumble } enter SAFE.SAFE_DETUMBLE
        exit do { exitSafeBase }
    }

    state SAFE_DETUMBLE {
        on safeCharge do { enterSafeCharge } enter SAFE.SAFE_CHARGE
        on safeCom do { enterSafeCom } enter SAFE.SAFE_COM
        on safeBase do { enterSafeBase } enter SAFE.SAFE_BASE
        exit do { exitSafeDetumble }
    }

    state SAFE_CHARGE {
        on safeDetumble do { enterSafeDetumble } enter SAFE.SAFE_DETUMBLE
        on safeCom do { enterSafeCom } enter SAFE.SAFE_COM
        on safeBase do { enterSafeBase } enter SAFE.SAFE_BASE
        exit do { exitSafeCharge }
    }

    state SAFE_COM {
        on safeCharge do { enterSafeCharge } enter SAFE.SAFE_CHARGE
        on safeDetumble do { enterSafeDetumble } enter SAFE.SAFE_DETUMBLE
        on safeReboot do { enterSafeReboot } enter SAFE.SAFE_REBOOT
        on safeBase do { enterSafeBase } enter SAFE.SAFE_BASE
        on charge if chargeIsSafe do { enterCharge } enter CHARGE
        exit do { exitSafeCom }
```

```
    }

    state SAFE_REBOOT {
        on safeBase do { enterSafeBase } enter SAFE.SAFE_BASE
        exit do { exitSafeReboot }
    }
}
```

References

[1] Jamie Chin, Roland Coelho, Justin Foley, Alicia Johnstone, Ryan Nugent, Dave Pignatelli, Savannah Pignatelli, Nikolaus Powell, and Jordi Puig-Suari. Cubesat 101: Basic concepts and processes for first-time cubesat developers. Technical report, California Polytechnic State University, San Luis Obispo (Cal Poly) CubeSat Systems Engineer Lab, 2017.

[2] The F´ Framework Team. F´: A Flight-Proven, Multi-Platform, Open-Source Flight Software Framework. URL `https://github.com/nasa/fprime`.

[3] Robert L Bocchino, Jeffrey W. Levison, and Michael D. Starch. Fpp: A modeling language for f prime. In *2022 IEEE Aerospace Conference (AERO)*, pages 1–15, 2022. doi: 10.1109/AERO53065.2022.9843754.

[4] ECSS-E-ST-70-11C Rev.1 Working Group. ECSS-E-ST-70-11C Rev.1: Space engineering and Space segment operability. Technical report, European Space Agency, 15 October 2025.

[5] The Linux Foundation. The Yocto Project: An open source collaboration project that helps developers create custom Linux-based systems regardless of the hardware architecture. URL `https://git.yoctoproject.org`.

[6] EPFL Spacecraft Team. CHESS Pathfinder 0 Satellite Project File. Technical report, École Polytechnique Fédérale de Lausanne, 8th December 2025.

[7] Stephanie Parker. The EPFL Spacecraft Team is launching EPFL back into space. Technical report, École Polytechnique Fédérale de Lausanne, 1st February 2023.

[8] Mohammed Eshaq, M. Sami Zitouni, Shadi Atalla, Saeed Al-Mansoori, and Malcolm Macdonald. Cubesat flight software: Insights and a case study. *Journal of Spacecraft and Rockets*, 62(4):1328–1345, 2025. doi: 10.2514/1.A35882. URL `https://doi.org/10.2514/1.A35882`.

[9] Ibtissam Latachi, Tajjeeddine Rachidi, Mohammed Karim, and Ahmed Hanafi. Reusable and reliable flight-control software for a fail-safe and cost-efficient cubesat mission: Design and implementation. *Aerospace*, 7(10), 2020. ISSN 2226-4310. doi: 10.3390/aerospace7100146. URL `https://www.mdpi.com/2226-4310/7/10/146`.

[10] M Doyle, A. Gloster, C. O'Toole, J. Mangan, R. Dunwoody, J. Thompson, and D. Sherwin. Flight software development for the eirsat-1 mission. In *Proceedings of the 3rd Symposium on Space Educational Activities*, page 157–161. University of Leicester, 2020. doi: 10.29311/2020.39. URL `http://dx.doi.org/10.29311/2020.39`.

[11] Olman Quiros Jimenez and Duncan D'Hemecourt. Development of a flight software framework for student cubesat missions. *Revista Tecnología en Marcha*, 12 2019. doi: 10.18845/tm.v32i8.4992.

[12] Koffi V. C. K. de Souza, Yassine Bouslimani, and Mohsen Ghribi. Flight software development for a cubesat application. *IEEE Journal on Miniaturization for Air and Space Systems*, 3(4):184–196, 2022. doi: 10.1109/JMASS.2022.3206713.

[13] Daniel Dvorak. *NASA Study on Flight Software Complexity*. doi: 10.2514/6.2009-1882. URL `https://arc.aiaa.org/doi/abs/10.2514/6.2009-1882`.

[14] Carlos E. Gonzalez, Camilo J. Rojas, Alexandre Bergel, and Marcos A. Diaz. An architecture-tracking approach to evaluate a modular and extensible flight software for cubesat nanosatellites. *IEEE Access*, 7: 126409–126429, 2019. doi: 10.1109/ACCESS.2019.2927931.

[15] Shaina Johl, E. Glenn Lightsey, Sean M. Horton, and Gokul R. Anandayuvaraj. A reusable command and data handling system for university cubesat missions. In *2014 IEEE Aerospace Conference*, pages 1–13, 2014. doi: 10.1109/AERO.2014.6836368.

[16] Timothy Canham. The mars ingenuity helicopter - a victory for open-source software. In *2022 IEEE Aerospace Conference (AERO)*, pages 01–11, 2022. doi: 10.1109/AERO53065.2022.9843438.

[17] The Clang Team. Clang-Tidy: A Clang-based C++ "linter" tool, . URL `https://clang.llvm.org/extra/clang-tidy/`.

[18] Charles Marsh. Ruff: An extremely fast Python linter and code formatter, written in Rust. URL https://github.com/astral-sh/ruff.

[19] Anthony Sottile and Ken Struys. pre-commit: A framework for managing and maintaining multi-language pre-commit hooks. URL https://github.com/pre-commit/pre-commit.

[20] The Clang Team. ClangFormat: A tool to format C/C++/Java/JavaScript/JSON/Objective-C/Protobuf/C code, . URL https://clang.llvm.org/docs/ClangFormat.html.