SSC20-P3-18

A Novel Approach to CubeSat Flight Software Development Using Robot Operating System (ROS)

Samuel Buckner, Carlos Carrasquillo, Marcus Elosegui, and Riccardo Bevilacqua
University of Florida Department of Mechanical & Aerospace Engineering
939 Center Dr, Gainesville, FL 32611; 239-565-3802
samuelcbuckner@ufl.edu

ABSTRACT

The large-scale development and deployment of small satellite systems in the modern aerospace sector motivates the parallel development of new flight software and avionics systems to meet the demands of modern spaceflight operations. Robot Operating System (ROS) provides an open-sourced and modular software framework that can be adapted for space-faring purposes in addition to the standard robotics operations that it is intended for.

At the University of Florida's Advanced Autonomous Multiple Spacecraft (ADAMUS) Laboratory, this framework, serving as the first flight-ready implementation of ROS in spacecraft flight software architecture, will be used in two upcoming CubeSat missions: (1) The Drag De-Orbit Device (D3) mission and (2) the PAssive Thermal Coating Observatory Operating in Low earth orbit (PATCOOL) mission. These missions will serve to validate the reusability of this software and the core functionality contained within, supporting tasks such as data link transmission, command processing, GNC (guidance, navigation & control), avionics system health diagnostics and soft/hard reboot instantiation. These capabilities draw upon the foundations of ROS in which tasks are handled by software modules that communicate through a language-agnostic messaging system resulting in an efficient multiprocessing architecture. Improvements and design considerations are also proposed in addition to suggestions for possible future use-cases of ROS in spaceflight software design.

Introduction to ROS

Contrary to its name, Robot Operating System (ROS) is not a standard operating system, rather it acts as a framework to establish communication between multiple software modules (henceforth referred to as **nodes**). These nodes communicate over a language-agnostic communications protocol referred to as **messages**, which consist simply of a list of data structure types and identifiers, such as `float64 time` or `Point32 points`. These messages are transmitted through **topics**, in which nodes can both publish and subscribe to any given topic or multitude of topics. **Services** are another feature of the ROS framework however are left outside of the scope of this paper. More background and information can be found through literature[1] and online.[2]

So why choose ROS for this application as a space-faring software framework? There are a number of reasons that can be reduced to the following:

- **Modularity:** Individual nodes can be easily swapped (such as those pertaining to simulation vs. sensed data feed)
- **Reusability:** Functionality can be tweaked and adjusted without disrupting the whole system, granting capability to support multiple missions and spaceflight configurations
- **Multi-lingualism:** Serialized/embedded and GNC software can be developed in C++, while most other basic functionality can be developed using Python (as well as several other languages)

Perhaps most importantly, however, ROS (as a free and open-source software) is backed by an engaged developer community with hundreds of software packages available to perform tasks such as computer vision, extended Kalman filtering (EKF) and spatial frame management.[1]

Drag De-Orbit Device (D3) Mission

The Drag De-Orbit Device (D3) CubeSat mission[3] aims to show how aerodynamic drag can be used as a means for orbital maneuvering, collision avoidance and de-orbit location (latitude/longitude) targeting, in addition to maximizing orbital decay to support Space Situational Awareness (SSA) efforts. This is facilitated through the deployment of

four booms that can be deployed and retracted to any intermediate length, allowing modulation of the aerodynamic drag experienced by the satellite while in low Earth orbit (LEO). The software and testing development timeline is laid out in the appendix in figure 5, targeting completion by the end of 2020, with a final launch date targeting no earlier than October 2021.

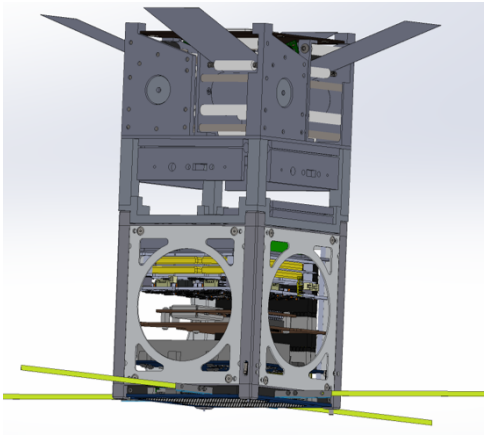

Figure 1: Full system model of the 2U D3 CubeSat with the drag de-orbit device on top and the avionics stack on bottom.

Mission Concept of Operations: The base concept of operations for the mission begins with deployment from an associated launch provider's payload module. Initialization will take place, including solar power collection, antenna deployment and telemetry collection. The D3 vehicle will then use an onboard magnetorquer with an enforced B-dot detumbling algorithm to stabilize the vehicle's attitude rates. The satellite will proceed to fly at maximum boom deployment for a majority of the mission timeline to ensure maximized orbital decay. This state will be maintained for as long as longitude targeting controllability can be feasibly maintained, at which point an external ground station will generate and uplink a guidance profile for the satellite to follow, parametrized by the ballistic coefficient (C_b). This coefficient has a linear correspondence to boom deployment length (of which all four booms will be deployed to equal lengths to maintain axial stabilization of the vehicle). An on-board guidance tracker will enforce small modifications to C_b values in response to GPS readings using a linear-quadratic-regular (LQR) control strategy to steer the vehicle towards the intended guidance profile in the wake of state error accumulation. The guidance profile will be periodically regenerated and uplinked to ensure accurate targeting leading up to atmospheric reentry. A detailed diagram visualizing these concept of operations can be seen in the appendix in figure 6.

Hardware: The flight software is under development to run on the 32-bit ARM processor aboard the BeagleBone Black. The BeagleBone Black was selected as the flight controller based on its capable processor, small form factor, and accessible I2C busses (of which it has three). The Clyde Space 3rd generation, 20Wh CubeSat battery and Electronic Power System (EPS) were selected to provide the assembly with a stable power source. A set of five solar panels manufactured by DHV Technology will actively recharge the battery to prologue the duration of the flight. The SkyFox Labs piNAV-NG L1 GPS receiver and the corresponding piPATCH-L1 antenna will actively report a position fix, deemed critical for guidance tracking. All uplinks and downlinks will be facilitated by the ISIS antenna system- the onboard UHF/VHF transceiver for all communication between the D3 and a ground station. An Invensense ICM-20948 inertial measurement unit (IMU) is also used to provide angular rates for attitude control with the magnetorquer.

Core Software Modules

Principal to discussion of the ROS framework are the core software modules that have been developed and packaged to run on the framework. Each software module described here can be considered to be a ROS node or collection of nodes (cluster) that can be assembled to run consecutively with a `roslaunch` XML-formatted file handled by a master `roscore` process. These processes also run alongside a local ROS parameter server (`rosparam`) for distribution of environment variables used at run-time. The overall architecture of these software modules can be represented with a standard ROS topological graph such as that in figure 2.

Finite-State Machine (FSM): Allows the software to exist in one of several states as shown in Table 1. Each state has a defined criteria for commands sent to other nodes while in a given state, along with a transition criteria for when the software should switch to a new state. For example, the Detumble state ends when the vehicle attitude rates fall below a certain threshold, and the GTrack state begins just before longitudinal targeting controllability will no longer be feasible. Note that not all states existing in the software are shown (just the fundamental ones), and that this node is not depicted in figure 2 for simplicity.

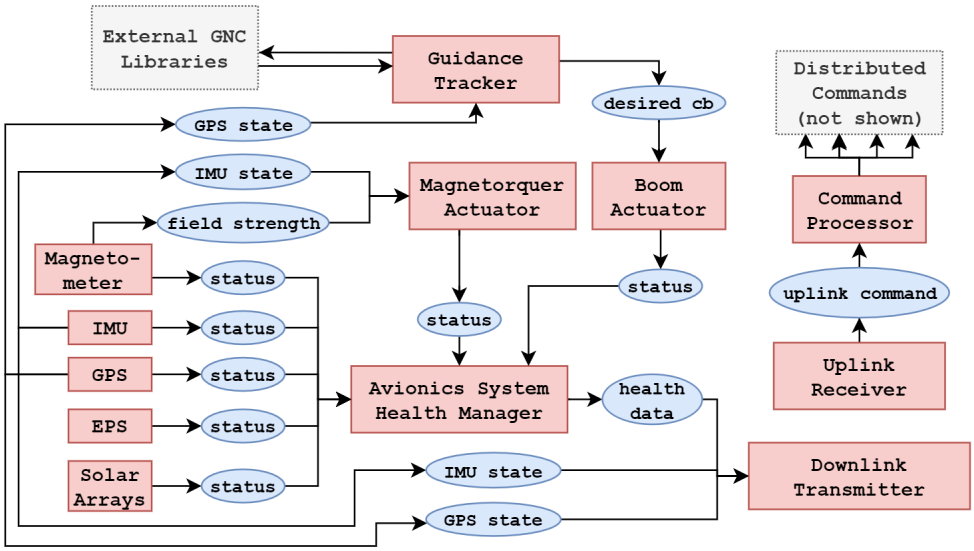

Figure 2: Conceptual diagram of the D3 software architecture with ROS nodes (red) and topics (blue) depicted. Connections involving the finite-state machine and command processor not shown for simplicity.

Table 1: Finite-State Machine States

Name	Description
Idle	Base functionality only
Initialize	Begin power distribution and diagnostics
Detumble	Magnetorquer control law in effect
GTrack	Guidance tracker control law in effect

Uplink Receiver: Assembles up-linked data packets containing two components: (1) An 8-bit integer identifying the type of command recieved and (2) raw data corresponding to the type of command sent assembled into ROS message format.

Command Processor: Receives assembled message from *Uplink Receiver*, identifies the command and correspondingly performs the prescribed protocol. The different types of commands are depicted in Table 2.

Specific notes on the functionality of each command are as follows:

- **Commands 1-2:** Involves killing all node processes and refreshing the `roscore` while maintaining memory of the previous system state before reboot.
- **Command 3:** Requires separate uplink of any file(s) to be updated and may require a soft reboot and recompilation with `catkin_make` depending on the type of file(s).
- **Commands 4-5:** Requests trajectory and/or diagnostics CSV files constructed by *Downlink Transmitter*.
- **Command 6:** Scrubs any accumulated and unnecessary files in the file-system.
- **Command 7:** Overrides *Boom Actuator* to command each boom to a custom deployment length for a prescribed amount of time.
- **Commands 8-10:** Updates parameters in the local ROS parameter server.

Table 2: Uplink Commands

Identifier	Description
1	Soft Reboot
2	Hard Reboot
3	Software Update
4	Request Telemetry
5	Request Diagnostics
6	Scrub File System
7	Override Boom Deployment
8	Update Detumble
9	Update Guidance Generation
10	Update Guidance Tracking

Avionics System Health Manager: Collects status data from on-board avionics components (such as voltage, current and power readings) and runs system-wide diagnostics to ensure all components are operating properly and not in risk of corruption. This includes unit-testing during power

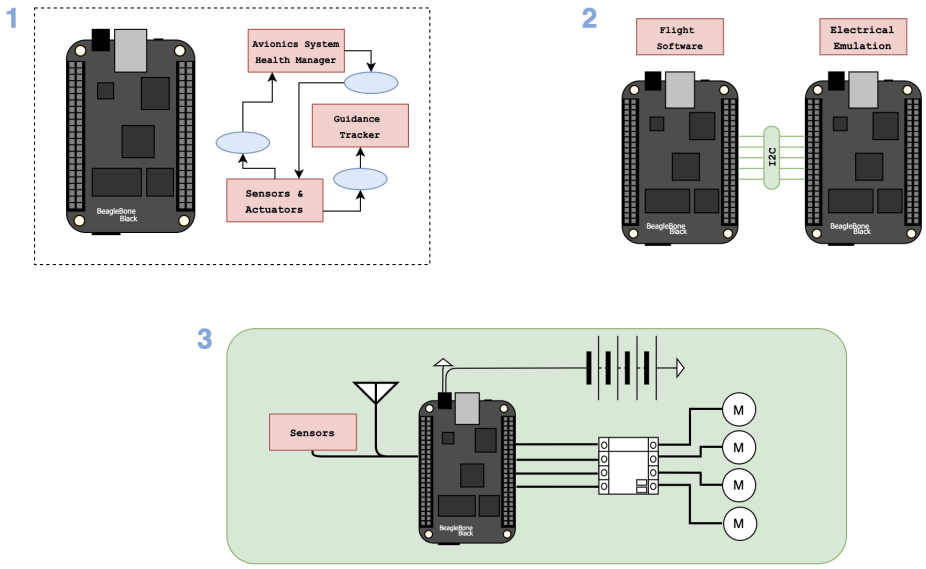

Figure 3: A very high-level overview of the three-layered approach used to test the D3's flight software. The (1). software-in-the-loop, (2) hardware-in-the-loop, and (3) FlatSat stages are depicted. Only select functionality was shown for clarity.

distribution and initialization. The components sending status data are manifested in the software as their own hardware nodes, such as the *Boom Actuator* and *Magnetorquer Actuator* (explicitly mentioned below) as well as the magnetometer, IMU, GPS, EPS and solar arrays.

Downlink Transmitter: Assembles message output from other nodes based on message field name and attribute into two CSV files for telemetry (such as GPS and IMU state measurement readings) and diagnostics (data accumulated by *Avionics System Health Manager*). These files are appended to at the same rate as node subscription, and a new file is started whenever a file downlink for the previous corresponding file has been successfully performed.

Guidance Tracker: Receives current GPS sensor state measurements, filters the sensor data through an EKF, and uses LQR control to determine the corresponding ballistic coefficient to remain on the desired guidance profile. Much of this functionality is built into separate external C++ libraries referenced by this node.

Boom Actuator: Converts desired ballistic coefficient message input from *Guidance Tracker* to a four-component array of floats between 0 and 1, where 0 corresponds to a fully-retracted (homed) boom and 1 corresponds to a fully-deployed boom. For all nominal cases, these components are all equal to maintain axial stabilization for the satellite, however this can be overridden in the prior-discussed *Uplink Commands*. This node also calls functionality to command each boom's DC motor to the desired number of wheel encoder steps.

Magnetorquer Actuator: Receives current magnetic field strength from the magnetometer and uses a B-dot law detumbling algorithm to calculate the corresponding torque commands required for detumble. Detumbling commands cease once IMU-sensed angular rates drop below a maximum allowable threshold.

Three-layered Testing Approach

As part of its development, the D3 flight software will undergo a comprehensive examination such that any bugs in the architecture and/or function are caught well prior to its maiden flight. A three-layered approach was deemed most effective to isolate bugs into one of three classes: the core ROS nodes, the telemetry transmission nodes, and the avionics communication nodes.

Software-in-the-Loop: Utilizes a singular BeagleBone Black running the entire flight software on one `roscore`. The software loop requires the defi-

nition of all message formats and the instantiation of all flight-critical nodes and topics. Simulated data will be generated and transmitted by each of the sensor nodes. As a requirement, unit tests must be built-in to all nodes. This development stage aims to identify and solve problems with the architecture of the software.

Hardware-in-the-Loop: Requires two BeagleBone Blacks running in parallel. One BeagleBone is responsible for running the D3 flight software with the inclusion of all telemetry nodes. A second BeagleBone responsible for emulating the data produced by the actual sensors and actuators and relaying it to the flight controller. All communication protocols (e.g. I2C, UART) implemented will be sensor-specific to simplify the transition into the next stage of development. This development stage aims to integrate the flight software with mission-critical telemetry software.

FlatSat: Utilizes all of the D3 CubeSat's avionics to perform a closed-loop, real-time examination of the flight software and all relevant sensors and actuators. The components are to be laid out on an anti-static mat to facilitate the process of probing all busses and power lines. This development stage aims to identify any bugs in the hardware and/or software consequential to the inclusion of the avionics. The D3 is not expected to progress from this stage until all transmitted data is accurate, reliable, and robust.

Once the D3 CubeSat has cleared all three stages of its developmental testing, no changes are to be made to the flight software unless all relevant testing stages are revisited.

PAssive Thermal Coating Observatory Operating in Low earth orbit (PATCOOL) Mission

The PATCOOL CubeSat mission[4] is a testbed for the performance of experimental cryogenic selective surface samples in LEO. Two samples with the experimental passive thermal coating are located in the payload housing structure at the head of the CubeSat. The intent of the experiment is to simulate the thermal coatings performance at an orbital distance of 1 AU from the Sun, thus the CubeSat's attitude must be controlled such that the payload is constantly zenith pointing, with respect to the Earth, so that the thermal loads from Earth's albedo are kept at a minimum. To further simulate this environmental intent, the structure of the CubeSat is designed in such a way as to minimize heat transfer to the samples. Temperature readings taken by thermistors within these samples will be taken periodically and transmitted down to a ground-station to append the thermal data to a database for further analysis.

A secondary objective of the PATCOOL CubeSat mission is to test a novel attitude determination and control system (ADCS) developed by the ADAMUS Laboratory. This ADCS utilizes a single modified D3 boom that works in conjunction with a tip mass located at the bottom of the CubeSat. When deployed, the tip mass creates a useful gravity gradient torque that can be used to maintain passive 3-axis stabilization, reducing the amount of power that on-board magnetorquers use for attitude control.

Flight Software Implementation

The PATCOOL software functionality can be thought of as a subset of the D3 ROS framework. With the GNC algorithms being generated and implemented by a commercial off-the-shelf ADCS, the *Guidance Tracker* and *Magnetorquer Actuator* modules, along with several uplink commands, are not necessary for PATCOOL's mission. The unnecessary modules can simply be disabled by removing those nodes from the `roslaunch files`.

Figure 4: Full system model of the 3U PATCOOL CubeSat with the gravity gradient boom deployed

The avionics on-board PATCOOL differ greatly from the avionics on the D3 CubeSat, with the ex-

ception of the command and data handling system as well as the electric power system. The embedded software to communicate with the avionics will be personalized to the avionics, but how the data is then handled by the ROS framework is reusable with some minor modifications. The general procedure for how the *Avionics System Health Manager* ensures proper operation of the electronic components is the same as for the D3 mission, with only the embedded software procedure needing to be changed. The reusability of the ROS framework is also apparent in the radio communications subsystem. The *Uplink Receiver*, *Downlink Transmitter* and *Command Processor* modules can all be adopted for PATCOOL with changes to the embedded software. The reuse of the D3 flight software saves a critical amount of software development time permitting for an overall quicker mission completion time.

Future Improvements and Considerations

Automated Testing

Developing a core flight software framework that can be leveraged by multiple projects has several challenges. Central to the advantage of a software framework is improved reliability. Integrating automated software testing into the flight software stack will improve the quality of future development, validate code changes and subsequently document the software design. As development expands it will be critical to establish an automated testing pipeline. Integrating ROS unit and integration testing scripts into the catkin build process will save time, isolate bugs and simplify the development process.

In addition to adding automated testing to the build and deployment pipeline, automated tests can be leveraged during flight operations to isolate and detect errors, as well as validate code changes during flight, preventing code corruption. Unit and integration tests can also be used to verify software health during operations. A brief description of these testing categories follows.

Unit Testing

Unit tests are function level software tests that verify functionality performs as designed. Future improvements might include automatically generating unit test scaffolding for new libraries and nodes.

Integrated Testing

Integration tests validate the function of a group of units when operated together. Future improvements might include standardized integration tests for core software modules, as well as automatically generating integration tests for new nodes. These tests would further standardize the expected interfaces between nodes.

Real-Time Operation

A common concern with the use of ROS relates to its inability to manage real-time operations, due to it inherently not being a real-time operating system (RTOS).[5] Tools have been developed, such as `ros_control` and RT-ROS,[6] to provide an integrated real-time task execution environment, which provide suitable functionality for the missions described in this paper. Regardless, for more dynamic systems this may not prove adequate, and other possibilities may need to be considered instead.

Comparisons to cFS and F-Prime

When discussing popular flight software frameworks, two alternatives come to mind: NASA Goddard Space Flight Center's core Flight System (cFS) and NASA Jet Propulsion Laboratory's F Prime framework.[7] ROS, as a software framework, falls somewhere between the two in terms of its capabilities:

- Like F Prime, ROS is designed for use with small-scale flight systems.
- Like cFS, ROS relies on a publish-subscribe scheme over `topics` with the added capability of `services`.
- Like cFS, ROS does not rely on code generation (which, in the case of F Prime, requires a commercial license with the MagicDraw software).

While the implementation of ROS as a spacecraft software framework as described in this paper cannot claim as much development, testing and validation as the aforementioned frameworks, it may be found suitable for small-scale missions with a desire for the modularity and ease-of-use that ROS is known to provide. Furthermore, for any system that may involve more traditional robotic capabilities or other popular tools (such as EKF and computer vision), the ease of integration and implementation of these publicly-available libraries may prove to save significant developmental effort in the long run.

Conclusion

This paper has presented Robot Operating System (ROS) for use as a flight software framework staged for validation aboard two CubeSat missions. Specific details regarding implementation in each of these missions, as well as improvements and considerations for future use of this framework, have also been covered. The use of ROS in this application presents a modular, flexible and easy-to-use architecture suitable for many different types of small-scale spaceflight systems.

Acknowledgements

The authors acknowledge other supporting members of the ADAMUS Laboratory and those on the D3 project team for their contributions, as well as Brian Hicks for consultation on inclusion of automated unit and integration testing.

References

[1] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[2] https://www.ros.org/.

[3] David Guglielmo, Sanny Omar, Riccardo Bevilacqua, Laurence Fineberg, Justin Treptow, Bradley Poffenberger, and Yusef Johnson. Drag deorbit device: a new standard reentry actuator for cubesats. *Journal of Spacecraft and Rockets*, 56(1):129–145, 2019.

[4] Kevin Bauer, Cindy Fortenberry, and Robert C Youngquist. Passive thermal coating observatory operating in low earth orbit (patcool). 2019.

[5] Sharath Chandra Amarawadi et al. Evaluation of ros and arduino controllers for the obdh subsystem of a cubesat, 2012.

[6] Hongxing Wei, Zhenzhou Shao, Zhen Huang, Renhai Chen, Yong Guan, Jindong Tan, and Zili Shao. Rt-ros: A real-time ros architecture on multi-core processors. *Future Generation Computer Systems*, 56:171–178, 2016.

[7] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F prime: an open-source framework for small-scale flight software systems. 2018.

[8] Sanny R Omar and Riccardo Bevilacqua. Hardware and gnc solutions for controlled spacecraft re-entry using aerodynamic drag. *Acta Astronautica*, 159:49–64, 2019.

Appendix

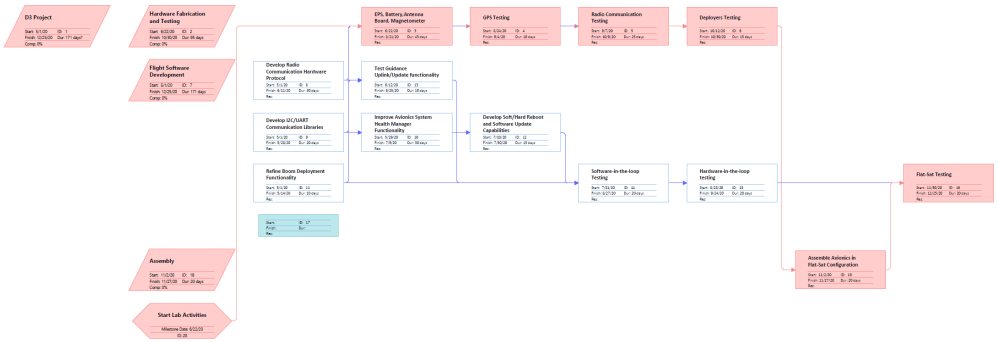

Figure 5: D3 mission timeline

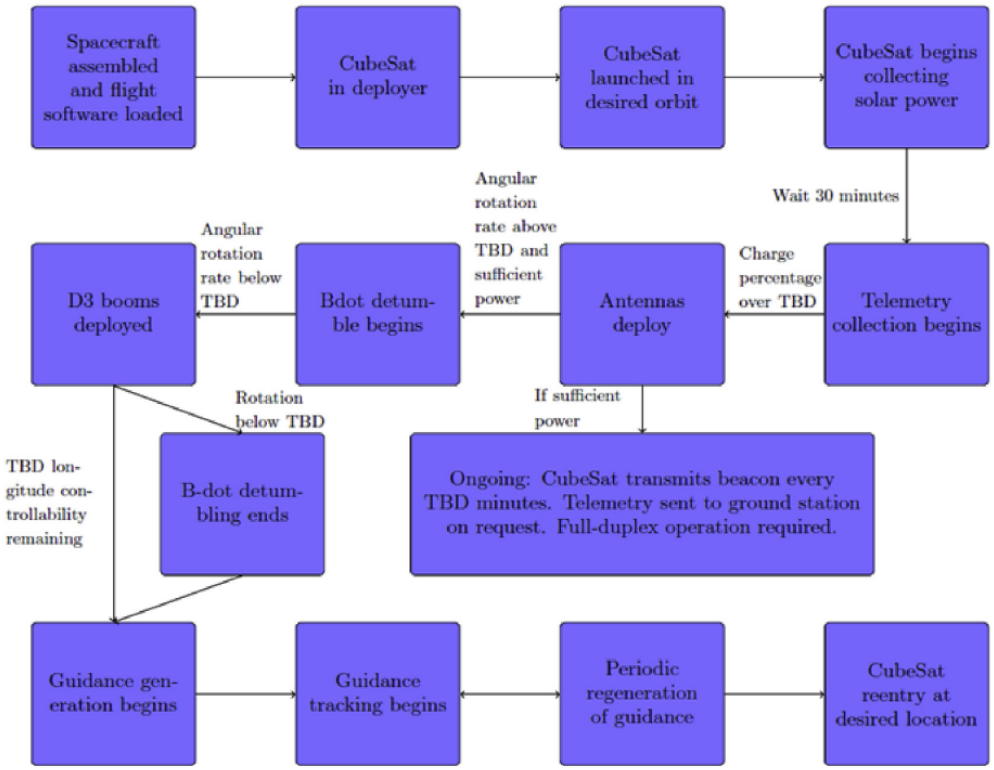

Figure 6: D3 mission concept of operations[8]