SSC21-I-01

Software-Enabled Smallsat Autonomy: Discussion with Examples

Timothy Woodbury, Austin Probe, Robert Effinger, John McGreevy, David Herceg, Matthew Ruschmann, Brett Carver, Timothy Esposito, and Graham Bryan
Emergent Space Technologies, Incorporated
7901 Sandy Spring Road Ste. 511, Laurel, MD 20707; 301-345-1535
tim.woodbury@emergentspace.com

ABSTRACT

Smallsat missions using cooperating constellations offer significant benefits compared to traditional space missions. These benefits include lower unit costs, better robustness to failures, and the ability to collect data in a distributed fashion. Significant commercial smallsat missions are active in low Earth orbit, and spacecraft operators have expressed interest in smallsat constellations operating both at higher altitudes and in proximity operations missions. Autonomy plays a significant role in extending smallsat missions to these more challenging domains. Autonomy in a broad sense refers to a spacecraft's or constellation's ability to operate independently of ground systems, and affects every part of a typical mission. For example, onboard processing of data can significantly reduce the frequency and expense of communications to a terrestrial ground station link. Onboard safety and health management is critical in proximity operations with fast dynamics, or in remote operations where offboard monitoring is available infrequently. Onboard monitoring of mission objectives enables remote operations and reduces the required operator workload.

Emergent Space Technologies has developed flight software products to enable future missions with greater autonomy. Navigator is a standalone application for cooperative absolute and relative navigation within a cluster of space vehicles. The Autopilot software suite enables routine orbit maintenance and satellite maneuvers to be monitored and executed onboard, increasing safety and reducing reliance on ground systems. Guardian is a suite of applications that enable fault detection, isolation, and recovery on modules within a distributed mission. The Cirrus cloud computing framework enables distributed computing tasks within a fleet of cooperating platforms, allowing complex data processing algorithms to be executed onboard and distributed among vehicles according to their computational availability. Finally, Commander is a set of applications for autonomous execution of a planned mission on a distributed group of platforms. Critically, Commander enables autonomous coordination of the actions of Navigator, Autopilot, Guardian, and Cirrus, providing a significantly greater level of autonomy than the suites provide individually. In this paper, we describe the capabilities of the flight software and demonstrate how coordination using Commander enables desired operator missions. The following missions are considered: (1) autonomous lunar injection; (2) rendezvous and proximity operations; (3) constellation intelligence, surveillance, and reconnaissance. Discussion is informed by use case diagrams and simulation results using Emergent's Ascent simulation environment.

INTRODUCTION

Distributed smallsat swarms and constellations are quickly becoming the norm in civil, military, and commercial space, driven by their low acquisition cost, robustness through redundancy, and distributed collection capabilities. These systems need to perform complex and coordinated maneuvers such as cooperative data collection, and rendezvous and proximity operations. Furthermore, module safety must be maintained by reacting appropriately to faults and external threats. As networks of proliferated space vehicles grow in size and push farther into deep space, current approaches to ground and FSW development will bottleneck deployment timelines, and mission operators will struggle with efficient command and control in response to dynamic events. To address problems of software design scalability and operator control inefficiency, mission design and ground operations can be elevated to the group level, rather than that of individual vehicles. To achieve this goal requires reliable onboard software capabilities that reliably address routine mission conditions.

Emergent Space Technologies, Incorporated (Emergent) has developed flight software (FSW) suites to support distributed onboard autonomy, with a focus on smallsat platforms. Emergent has a series of subsystem-level app suites that enable various mission-critical functions. These include Navigator, which performs absolute and relative navigation in a cluster; Autopilot, which plans optimized maneuvers and performs collision detection; Guardian, which performs fault detection, isolation, and recovery (FDIR) at the module and swarm level; and Cirrus, a space-based cloud computing framework.

Emergent's Commander autonomy software suite coordinates and manages these app suites during routine operations. Commander facilitates a clear division of responsibility between the operator, mission-level, agent-level, and task-level execution services, and focuses on technologies and algorithms that enable transparency, ease of understanding, and supervisory control over mission execution. Commander software can be run on-board or from the ground, with the operator in-the-loop or on-the-loop, thereby enabling an evolutionary approach to trusted autonomy. The operator interacts with Commander by issuing commands, monitoring telemetry, and uploading new plans.

The paper is structured as follows. First, we describe Emergent's flight software capabilities for multi-satellite autonomy. Next, we describe how those capabilities are orchestrated via the Commander FSW suite for distributed autonomous mission plan execution. Then, we explain Commander's core applications, plan structure, and integration approach with external software, such as satellite buses and payloads. Finally, we share use cases and simulation results for several relevant multi-satellite mission scenario demonstrations.

FLIGHT SOFTWARE CAPABILITIES

Emergent has developed flight software enabling autonomous spacecraft operations. Each flight software suite provides specific capabilities and can be related to a typical robotic software subsystem. Each suite is an enabling technology for a future fully autonomous spacecraft platform. On its own, each suite is independent and requires external information to respond appropriately to changing mission conditions and objectives. When flying on demonstration missions, that external information has come directly from ground commands. By leveraging the Commander software discussed in the next section, the spacecraft platform can be a source of these commands, allowing complex behaviors to be initiated and managed with minimal input from ground operators.

In this section, we discuss the FSW suites enabling particular subsystems. There are four such suites: Navigator, Autopilot, Guardian, and Cirrus. Navigator provides relative and absolute navigation capabilities for a cluster of spacecraft. Autopilot provides cluster management functions, analogous to cooperative motion planning in robotics. Guardian provides FDIR capabilities. Cirrus provides cloud computing capabilities that can be leveraged to execute advanced algorithms in situ, such as machine learning perception algorithms. This section briefly summarizes each flight software suite.

Navigator

Navigator provides estimation capabilities within a cluster of cooperating satellites. Originally developed for DARPA's System F6 program, Navigator fuses GPS and crosslink range measurements to provide state and covariance information about cluster assets[1]. Navigator can perform both absolute and relative state estimation, depending on mission needs and available sensors. Navigator leverages an onboard Extended Kalman Filter (EKF) for nonlinear state estimation and can execute up to four independent filters in parallel. Each filter can be individually configured for different state and measurement vectors, and configuration can be performed before or during runtime. The ability to execute multiple independent filters enables health checks and improves robustness to sensor failures, as discussed later in this section. By fusing information from cluster spacecraft, Navigator reduces the need to carry redundant backup GPS receiver components.

Navigator provides estimation capabilities for the following common navigation states, depending on the available sensors:

- Absolute position and velocity
- Relative position and velocity
- GPS clock bias and drift
- Relative range sensor bias
- Unmodelled accelerations

Navigator can associate state elements with each cluster module to fuse sensor measurements coming from multiple modules. Navigator also supports Consider Kalman Filtering of states. Consider filtering enables parametric uncertainty to be included in covariance calculations without explicitly estimating the uncertain states, potentially a large computational savings[2].

In addition to state estimation, Navigator monitors the health of each filter by performing various operations. Navigator routinely performs various sanity checks that can downgrade the health of a given filter (e.g., verifying positive definiteness of the covariance matrix). Innovations residuals are pre-processed before updating the state vector. If the normalized residual exceeds a configurable bound, the measurement is rejected. Additionally, a normalized innovation square (NIS) metric is used to detect biased innovations, which can result from faulty sensor measurements. If the NIS metric for a series of time steps exceeds a statistical threshold, the health of the filter is downgraded.

By executing multiple independent filters in parallel, Navigator enables the published state to be adjusted automatically with the filter's health. Navigator is configured with a relative priority for each executing filter, allowing developers to specify a "default" filter if all filters are healthy. The published state is prioritized based on the health of a filter, and then by its priority.

Navigator executes as an independent application, but it has integrations with other products that can significantly improve overall mission performance. Navigator and Autopilot were originally developed as a common software suite, and many of Autopilot's advanced features require input from Navigator or an equivalent estimator. Additionally, the Guardian FDIR product contains the Navigation Monitor application, which performs more advanced GPS integrity checks that can identify additional sources of error.

Autopilot

The Autopilot software suite provides semi-autonomous spacecraft guidance and control for a single spacecraft or groups of proximate spacecraft called clusters. The suite provides capabilities through a service-oriented architecture, which can be accessed by commands coming from a ground operator or from autonomy software like Commander[3]. This section discusses the primary flight software services provided by Autopilot. These services are as follow: Cluster Flight Manager (CFM), Orbit Maintenance Service (OMS), and Maneuver Planning Service (MPS).

CFM is designed as a single interface between Autopilot and the outside world. CFM orchestrates Autopilot's activities by providing statefulness and persistent memory. CFM manages periodic activities including reconfiguring the relative positions of cluster modules, station-keeping, and checking the probability of collision between objects in the cluster. CFM also receives maneuver plans from MPS and manages maneuvers throughout their lifecycle. MPS also manages the cluster inventory, allowing modules to be added or removed from the cluster dynamically during operation.

OMS provides computations relating to closed-loop cluster activities as services; these include reconfiguring, station-keeping, and probability of collision monitoring. OMS determines whether a predicted trajectory satisfies target orbit constraints. OMS can request new maneuver plans from MPS to support station-keeping activities. OMS is also responsible for validating candidate maneuver plans before distributing them to CFM. Finally, OMS performs probability of collision checks both as a service and as part of maneuver plan validation.

MPS is implemented as a stateless service for computing maneuver plans. A maneuver plan is a series of one or more maneuvers for one or more modules to achieve a target orbit(s) within a specified time window. The service-based design allows a requestor to specify maneuver goals, constraints, and design parameters, and responds with optimized maneuver plans. MPS leverages simulated annealing (SA) to obtain

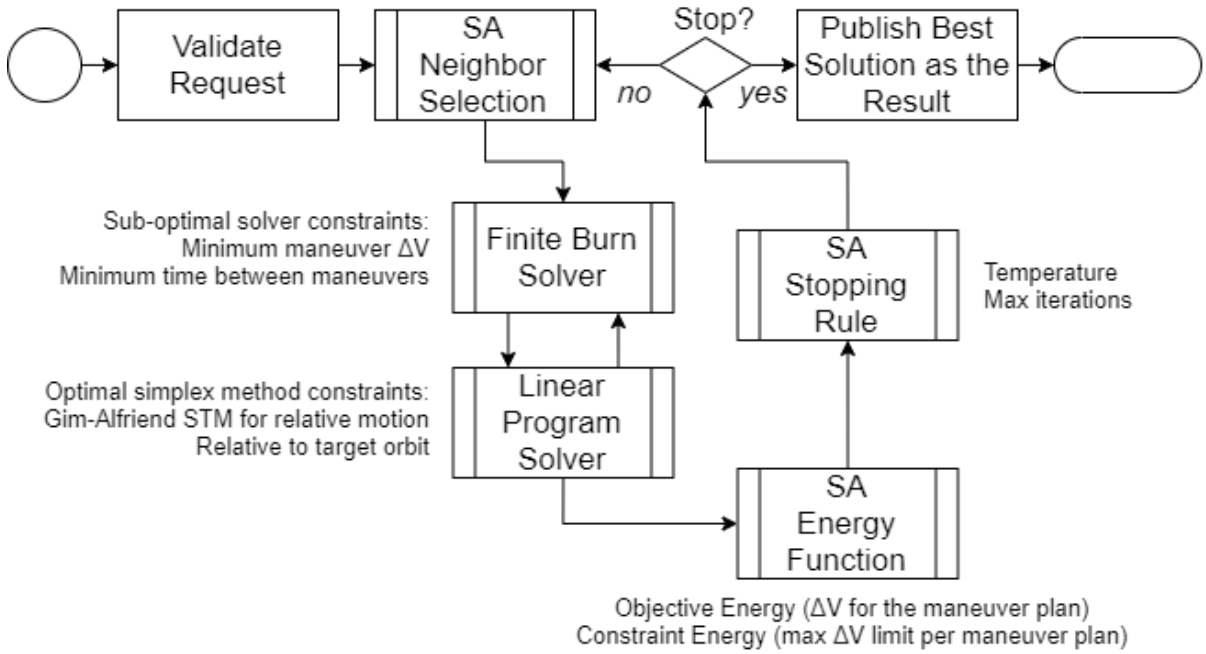

Figure 1: Autopilot's Maneuver Planning Service planning algorithm

operationally useful maneuver plans in a constrained amount of processing time. The process requires multiple steps, each of which is responsible for checking different constraints, as summarized in Figure 1. SA is used to search the maneuver planning space for candidate solutions. A candidate solution from the SA algorithm is passed to a finite burn solver, which uses the Gim-Alfriend state transition matrix (STM) to solve a linear program (LP) and convert the SA output into a sequence of finite burns. The resulting finite maneuver's cost is weighted by total maneuver cost and constrained by a maximum maneuver magnitude. The process repeats until an SA iteration limit is reached[4].

Guardian

Guardian provides fault management capabilities within multi-spacecraft mission architectures. Guardian's primary function is to monitor GPS navigation, inter-spacecraft comms, and propulsion. As with the other products, most of the software is designed using a service-oriented architecture. Additionally, Guardian software uses a hierarchical design with clearly defined levels, each having well-defined interfaces and functionality. This hierarchy is summarized in Table 1. The lowest level is the subsystem level, which is primarily responsible for executing recovery actions from the higher levels. The system and cluster levels perform detection and isolation actions at the appropriate levels within the cluster. Diagnosis and Recovery sit above those levels. Diagnosis is responsible for fusing health reports, and Recovery monitors the health reports and commands appropriate recovery actions[5]. The "Supervisor" level provides support when the recovery action is unclear or not covered by mission programming; the Supervisor is a generic representation that could represent information coming from external software, like Commander, or commands from a human operator.

Guardian contains software from the System Level to the Recovery level of the hierarchy. Subsystem Level functionality is provided by other software, such as Autopilot, which provides appropriate functionality for control of a guidance, navigation, and control subsystem. Guardian is closely integrated with Autopilot and Navigator, and some of its system level applications have additional capabilities when deployed with those apps. The remainder of this section discusses the primary software services within Guardian. These are Navigation Monitor, Thrust Monitor, Cluster Monitor, Diagnosis, and Recovery.

Navigation Monitor is a system-level application for health of GPS measurements. Navigation Monitor implements Receiver Autonomous Integrity Monitoring (RAIM) for processing GPS pseudorange measurements. Additionally, the app integrates with Navigator to determine if GPS or relative range is being used for localization, since some GPS health checking is already performed in Navigator. RAIM is a well-known algorithm that detects statistical outliers in GPS pseudoranges[6]. RAIM provides Navigation Monitor additional health monitoring beyond the statistical checks performed in Navigator.

Thrust Monitor uses a probabilistic approach to detect errors in the onboard thrusters. Thrust Monitor compares accelerometer measurements against expected thrust accelerations during maneuvers. The app also evaluates configurable fault modes, such as loss of effectiveness of a thruster, that enable more granular diagnosis of detected faults. A maximum likelihood-based algorithm originally presented by Wilson et al is used for detecting significant changes in thrust relative to expectations[7]. Thrust Monitor has shown an ability to detect failures that it is not specifically configured for, although the ability to accurately diagnose and isolate the problem will necessarily be degraded.

Cluster Monitor provides cluster-level services by fusing data from multiple managed modules. Cluster Monitor primarily provides navigation health checking by exploiting measurements from multiple modules. Two algorithms are used: Filter/Range Parity (FLT-PR) and Receiver Autonomous Integrity Monitoring Augmented with Relative Range Measurements (RAIM-RELRNG). FLT-RP compares navigation range residuals against relative range measurements using a χ^2 statistical test. RAIM-RELRNG fuses the GPS ranges from all

Table 1: Levels of Guardian Hierarchy

Level of Hierarchy	Description
1 Subsystem Level	Monitors diagnosed health and commands recovery actions
2 System Level	Detection and isolation components monitoring data with a very high guarantee of arrival
3 Cluster Level	Detection and isolation components monitoring data subject to transmission outages
4 Diagnosis	Fuses health reports into a single diagnosis of cluster health
5 Recovery	Monitors diagnosed health and commands recovery actions
6 Supervisor	Provides support for health conditions that have ambiguous causes that prevent safe automatic recovery

spacecraft with measured relative ranges to generate a nominally Gaussian parity vector. A statistical test is performed on the parity vector to detect non-Gaussian distributions, which indicate the presence of a fault. These additional checks can detect faults that are impossible to detect with a single module, improving the robustness of navigation health monitoring in the cluster.

Diagnosis is a highly configurable application for fusing health reports from the system and cluster levels into an overall diagnosis of cluster health. Diagnosis is implemented in the Lua scripting language and contains two primary scripts. First is a mission script that configures the application and is highly tailored. Second is a library of functions for fusing health messages. The library provides voting rules, which resolve conflicts among sources of information that disagree, such as a fault that is detected at the cluster level but not the system level. The library also provides conjecture rules, which provide the logic needed to isolate a faulty component when the raw health information indicates multiple candidate faults. The mission script configures how the Diagnosis library should be applied to a particular mission and its available health data.

Recovery provides functionality for responding to a detected fault and is highly configurable for each mission. Recovery responds to faults by sending software commands to other applications. Recovery's primary components are a recovery action table, and configurable state machines for recovery commands. The recovery action table maps identified faults to known recovery actions that should be taken. The configurable state machines are implemented to provide persistent memory for faults that require multiple steps or conditional behavior; e.g., Recovery might respond to a navigation error by power cycling a GPS receiver and waiting for an updated diagnosis to determine if the fault is resolved.

Overall, the Guardian apps provide configurable and flexible tools for FDIR within a cluster of spacecraft. The last app suite discussed in this section is Cirrus, which provides flexible space-based cloud computing capabilities.

Cirrus

The Cirrus flight software product performs onboard payload processing within a trusted, distributed network of space vehicles. This enables spacecraft to efficiently process data in situ without downlinking measurements to ground stations, which reduces bandwidth needs and makes actionable outcomes available more quickly to end users. The primary goal of Cirrus is to mimic terrestrial cloud computing, which enables an end user to request a computational task and get a result without ever being aware of the platform(s) used for performing the computation. Similarly, Cirrus enables computationally expensive tasks to be distributed to available computing nodes automatically and efficiently, without guidance from an end user. To achieve this goal, Cirrus deploys three core services: the Network Service, the Compute Service, and the Storage Service.

The Cirrus Network Service is responsible for connecting compute nodes and maintaining addressing information about available nodes. Its main features are network topology discovery, path-agnostic message routing, and store-and-forward messaging with an eviction policy. These capabilities provide the basic infrastructure that enables the Compute Service.

The Cirrus Compute Service provides functionality to monitor computational resources and execute tasks. The software can respond to queries about its host's capabilities, schedule tasks, perform task monitoring and control, and prioritize schedule of tasks. The results of a completed task are automatically packaged and sent to an address specified by the task creator. Similarly, the software monitors performance statistics such as RAM, CPU, and disk usage during task execution, and can report these to a specified monitoring address.

The Cirrus Storage Service provides for short- and long-term storage of files. It provides a file- and block-based API for managing stored artifacts. The service operates on a reservation and leasing structure to provide flexibility for mission planning or long durations without downlink communications. Eviction policies ensure that both reservations and leases are bounded and not indefinite. The storage API supports both file and block-based interactions, enabling files to be broken up and distributed across multiple nodes if no single node has enough memory available. Retrieval functionality includes retrieving file metadata, data blocks, and full file contents. All retrieval functionality can be performed for a single file or multiple files associated via a Cirrus global task ID assigned by the Asset Scheduler.

By leveraging the Network, Compute, and Storage Services, Cirrus enables end users to request computational tasks and retrieve results without regard to the underlying computer hardware. This allows expensive tasks to be distributed among trusted space assets, enabling new space capabilities, such as in situ processing of sensor measurements using machine learning algorithms.

Each of the software suites discussed in this section provides a particular capability, and implement a service-oriented design that enable those capabilities to be leveraged by external commands. In flight demonstration missions, those commands have come

from ground operators. The Commander autonomy software suite, discussed in the next section, provides a reconfigurable plan executor that can be used to manage the actions of other app suites as subsystems.

COORDINATION VIA COMMANDER

The FSW suites described in the previous section each provide a distinct set of capabilities relating to a mission subsystem. Each suite is implemented following a server-client model. This design enables flexible deployment on different buses, such that the FSW capabilities can be triggered by ground commands or external flight software as missions require. While the FSW suites provide distinct capabilities that enable mission success, they lack an overall framework for execution of a planned mission. The Commander FSW suite provides a framework for distributed autonomous execution of a mission plan. It is designed to execute finite state machines and integrate with arbitrary external applications based on a server-client model. The Commander FSW is also the logical interface between ground controllers and the flight software. Instead of monitoring and controlling individual apps, Commander correlates status information and can dispatch commands to managed apps in response to ground commands. In this section, we describe the Commander FSW suite, the core applications, the structure of a Commander Plan, and finally our flexible integration with external flight software.

Core Applications

Commander consists of three core apps with clearly defined roles and responsibilities. Mission Manager (MM) is a coordinator application that is responsible for receiving updates from managed platforms and ground commands. Execution Manager (EM) is the platform-level plan executor, and is responsible for monitoring the health of the platform and dispatching commands to managed apps. Timeline Manager (TM) is a queuing application responsible for task deconfliction. MM is deployed in a one-to-many fashion on autonomous platforms, while each platform executes an instance of EM and TM. One MM is not expected to coordinate all platforms in a fleet; for example, spatially dispersed fleets may have several active MMs due to communications constraints. The overall effect is a hierarchy where a ground operator(s) interfaces with one or more MM instances, each of which coordinates several platforms. This hierarchy is summarized in Figure 2 for three cooperating Commander instances. In this diagram, the arrows indicate the ideal flow of overriding commands (ground commands can supersede Mission Manager actions, Mission Manager can supersede Execution Manager). In practice, it is likely that ground operators would retain some ability to directly override Execution Manager activities for safety.

Plan Executors (Execution and Mission Manager)

EM and MM are derived implementations of a common software structure that is referred to as a Plan Executor. A "Plan" describes the state of the system should change in response to received telemetry, and what message(s) should be sent in response. We discuss Plans in more detail later in the paper. A plan is a set of software objects that follow a proscribed interface used by an executor. The primary software loop of an executor is outlined in Figure 3. The executor loads a binary file representing a state machine at startup. All state transitions are triggered by received messages coming from a message bus. Messages are processed by a TelemetryProcessor. The TelemetryProcessor is programmed with the message types and conditions associated with events, and it buffers event IDs in a TelemetryBuffer. The executor checks the TelemetryBuffer for event IDs. When an event occurs, a state transition is triggered. External messages can be triggered by three related occurrences: (1) the event itself; (2) exit of the current state; (3) entry into the new state. Messages are published to the external message bus.

Typically, EM sends TaskRequest messages to TM and monitors the execution of the task based on TM's published messages. Similarly, MM primarily sends

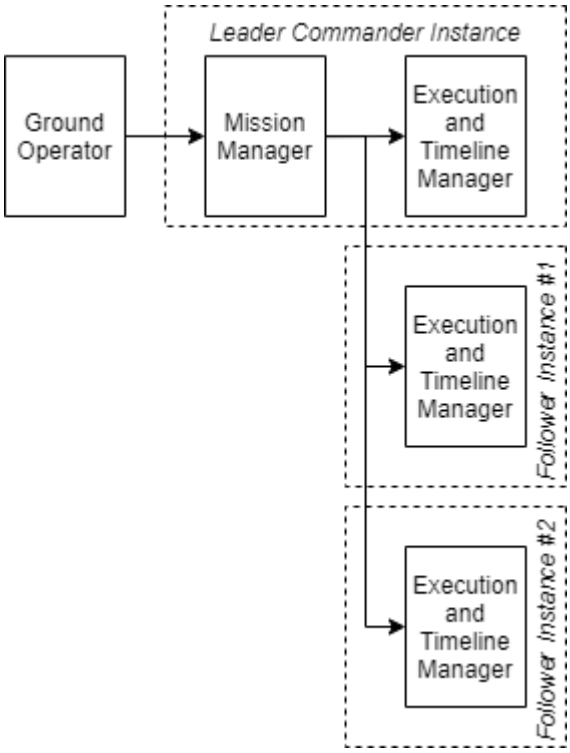

Figure 2: Commander operational hierarchy

messages that are intended for one or more managed EM instances. However, the messages to be sent are specified in the state machine object loaded by an executor. This gives mission designers considerable flexibility in the types of messages that can be published, and allows an individual app (MM or EM) to be tested or flown in the absence of the other core apps.

Timeline Manager

TM implements a timeline of tasks to execute based on task resources, priority, and execution state. TM interfaces with two "classes" of external applications: planners and workers. Planners generate tasks that need to be done and expect updates on task status. Workers accept tasks to be done and return status messages indicating progress. The message interface between TM and external software is shown in Figure 4. TM acts as a server to planners, publishing a TaskResponse message in reaction to a TaskRequest from a planner. The TaskRequest contains message data identifying a message that should be sent to a worker to begin the task. TM has a client relationship with workers. TM publishes an initial service request using the TaskRequest data, and expects responses from the worker indicating that the task has been started, and that the task has been completed. TM publishes additional messages to notify the planner of the change in task status, and the planner can also request task status updates from TM.

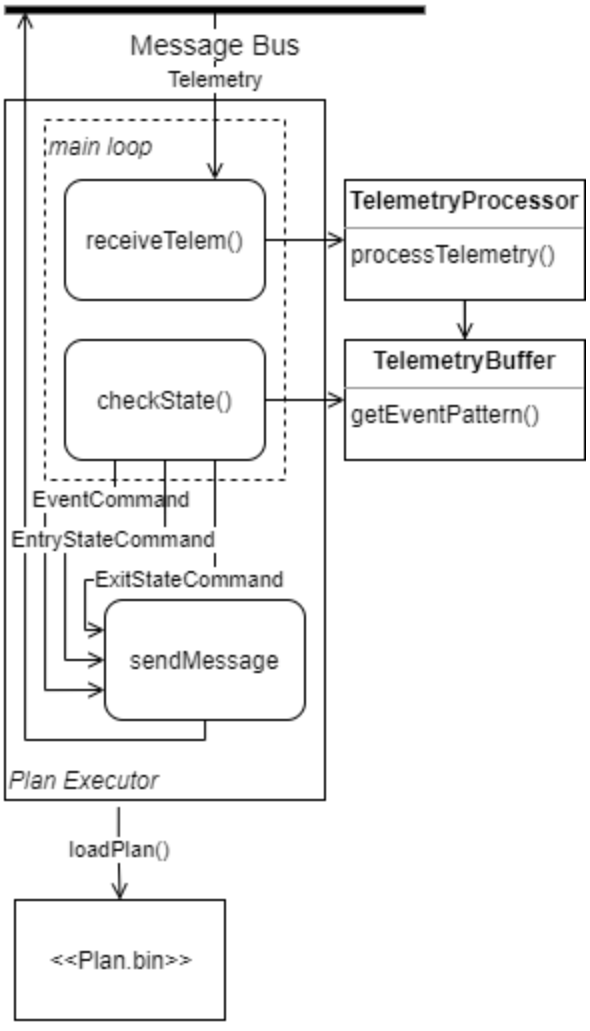

Figure 3: Plan executor core processing loop

TM implements a timeline of tasks based on provided priority and resource information. Resources are integers identifying hardware or software assets that are required for a task, and priority is an indication of relative importance. If a valid TaskRequest is received by TM, it is added to a running list. If the start window of a list task is reached and there are no already-queued tasks using the same resource, the task is queued and an associated service request is sent to a worker. The task remains active until the task either is completed by the worker or times out. The task is blocking for all other tasks with the same resource, regardless of priority. For example, consider Figure 5, showing the time windows of three tasks in the TM list. Task A will be immediately queued at t_1. Task B will not be queued unless Task A advances to the completed state, because resource 1 is blocked by A. If Task A completes before the end of Task B's window, then Task B will be queued. Task C will be queued immediately at t_2 because no other tasks use resource 2.

Plan Structure

A plan defines how a particular executor should behave in a particular mission. A plan consists of several parts, most of which are customized by mission designers.

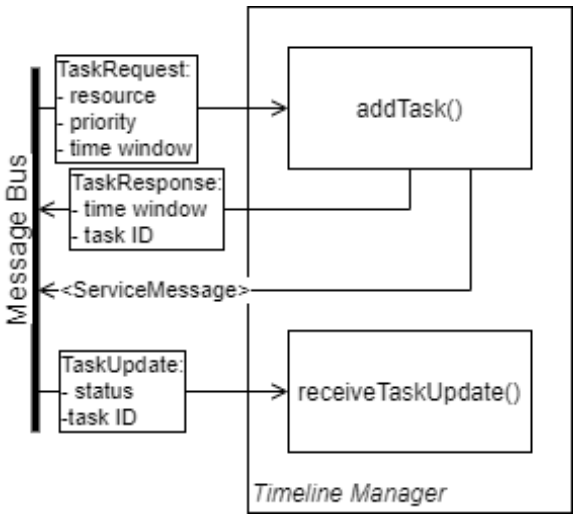

Figure 4: Timeline Manager communication diagram

Overall, a plan describes what messages should be sent by the executor in response to incoming messages. Messages are the only possible interactions with the outside "world," and a plan is essentially a message interface definition wrapped around a state machine.

Figure 6 shows the main software objects of a plan, along with the flow of information through the components. The TelemetryProcessor defines the message type and associated values that trigger events in the state machine. This object must be customized by the mission designer to specify the message conditions associated with events. Event IDs are stored in a buffer, which is checked during execution of the state machine. The TelemetryProcessor may also store data in a received message in a ValueStore, which is a generic memory object. The ValueStore memory is accessible to both incoming and outgoing messages, allowing outgoing messages to be customized on the fly in response to incoming telemetry. E.g., incoming messages might specify the time and direction of a planned maneuver. The state machine specifies state transitions in response to events in the TelemetryBuffer. State transitions and event occurrences can be associated with a message that should be sent. That message is preprocessed by a CommandGenerator, which is the counterpart of a TelemetryProcessor for outgoing messages. A CommandGenerator specifies how messages should be populated using data in the ValueStore. Each of these software components, with the exception of the TelemetryBuffer, follows an interface definition and is customized for a particular mission. This provides flexibility to change the behavior of the autonomy software for a particular application. This completes the discussion of a Commander plan.

Integration with External Software

Commander is designed to integrate with external apps, using the model that external software can be accessed via service requests. The TelemetryProcessor can be linked against an external library to receive and parse messages in a format unknown to the core Commander software. In the same way, the CommandGenerator can populate and send messages. This allows Commander to send service requests and process responses from an outside, "black-box" application, as long as it has a well-defined service interface. While this software design is intended primarily for Commander acting as a client to external services, the same structure can also be used to access Commander from an external client. This would allow, e.g., an outside bus manager application to manage Commander using its native interface, rather than having to comply with Commander's interface. Overall, the design of Commander streamlines its integration with external software by providing a structure for defining external interfaces with which Commander should comply.

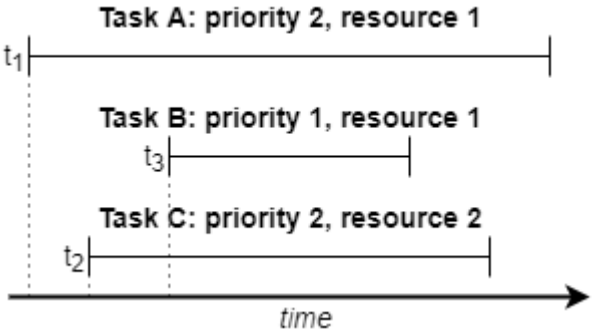

Figure 5: Timeline Manager timeline diagram

This completes the discussion of the Commander flight software. The next section describes the example integration missions demonstrating the autonomy software applications working together.

EXAMPLE MISSIONS

To demonstrate the use of Commander to manage FSW and integration with other FSW suites we describe characteristic deployments of Commander. The first is a cislunar demonstration in which Commander manages lunar insertion of an autonomous spacecraft. The second integrates Commander with Navigator and Autopilot to manage an autonomous Rendezvous and Proximity Operation (RPO) scenario. The third deployment integrates Commander to manage data acquisition and processing inside Cirrus for an Intelligence,

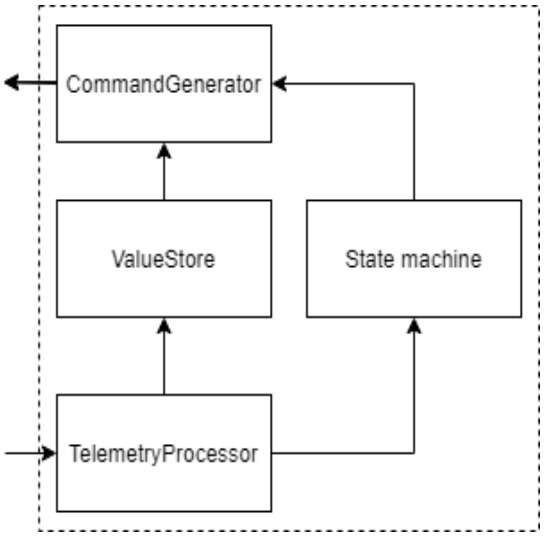

Figure 6: Plan software components and data flow

Surveillance, and Reconnaissance (ISR) scenario. These FSW scenarios are demonstrated using the Ascent simulation environment, which is described next.

Ascent Simulation

Ascent combines a distributed high-fidelity physics-based dynamic simulation with a highly scalable message bus for integrating FSW and other external components for testing. The dynamic simulation models spacecraft dynamics and hardware components to provide realistic inputs to the FSW being evaluated. This simulation is deployed in a containerized environment to maximize scalability and portability. Ascent has demonstrated the capability to scale to provide FSW simulation for hundreds of vehicles making it ideal for FSW testing for distributed space missions. Ascent is used to generate the simulated results for the missions described in this section.

Cislunar Mission

The cislunar mission emulates one phase of a generic lunar science mission with a single spacecraft. This scenario is a minimal demonstration of the Commander FSW showing multiple state transitions and interprocess communication. The autonomous spacecraft is initially on a parabolic polar orbit around the Moon, and its goal is to perform an injection burn at periapsis.

The Commander mission artifacts are summarized in Figures 7 and 8. MM and EM are programmed with linear state progression that is triggered by simple telemetry conditions. Figure 7 shows the mission states, and the associated events that trigger state transitions.

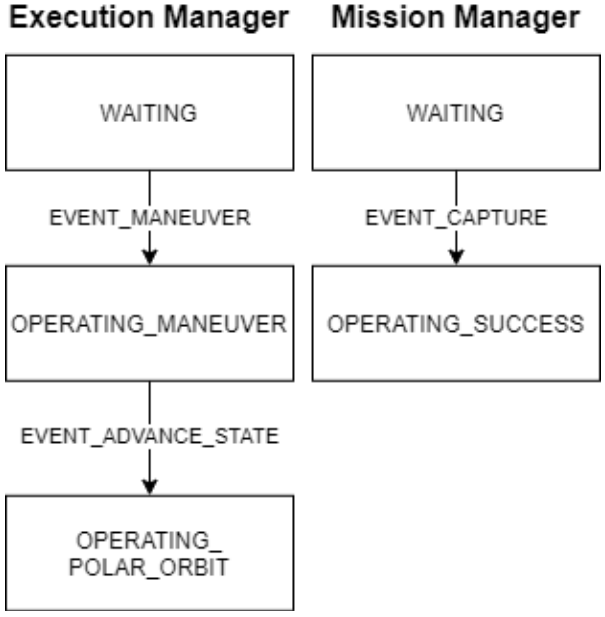

Figure 7: Cislunar Mission State Diagrams

Figure 8 shows the EM telemetry logic that causes various events to be buffered. MM uses a similar structure, but processes a Maneuver Report message from Ascent to trigger the EVENT_CAPTURE condition. In the mission, both executors are initially in a waiting state. Once EM receives a navigation message indicating the spacecraft is within a target bound of maneuver coordinates, the EM transitions to the OPERATING_MANEUVER state. Associated with this state transition is a Task Request message that specifies a Maneuver that should be sent to the Ascent simulation. The Task Request is processed by TM, and the Maneuver command is sent. This causes the simulated spacecraft to execute a maneuver and send a Maneuver Report if the maneuver is successful. The Maneuver Report is received by MM, triggering EVENT_CAPTURE. This transition is associated with an "Advance State" message sent by MM, instructing the EM to advance to the terminal state. EM receives the message, checks it, and advances to the OPERATING_POLAR_ORBIT state. This is a notional state that corresponds to the beginning of the nominal science mission.

Figure 9 shows a visualization of the Ascent simulation, with the spacecraft in motion around the Moon. This mission is simple and intended to demonstrate a concrete implementation of the Commander FSW. Some simple extensions can make the mission plan more robust and realistic. For example, when the maneuver is complete, Commander can validate that the expected orbit is reached to an appropriate tolerance. If not, it can leverage Autopilot to request an additional maneuver to reach the target orbit. In the same way, thruster health

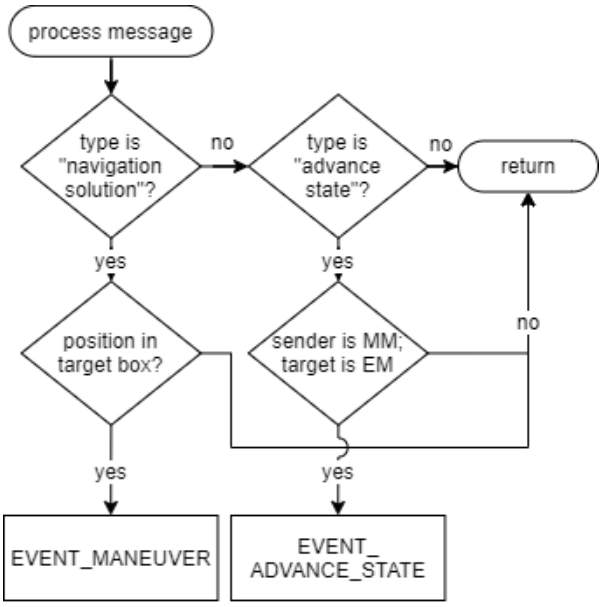

Figure 8: Cislunar Mission Telemetry Logic for Execution Manager

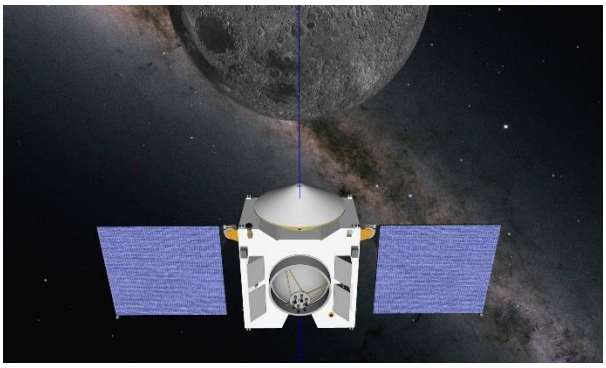

Figure 9: Visualization of Cislunar Mission in Progress

information from Guardian can be leveraged before the maneuver to check that the bus is healthy. If the thrust available is significantly degraded from the design thrust, Commander could choose to remain on or return to the initial parabolic orbit to attempt a return to Earth.

Rendezvous and Proximity Operations

Upcoming missions, such as Mars Sample Return (MSR), depend on successful RPO and On-orbit Service and. Manufacturing in environments where ground communications are unreliable. On-board autonomy is needed to enable safe operations with rapid detection and response to failures. Emergent's Navigator and Autopilot software suites provide the localization and autonomous maneuvering capabilities required for these operations. As currently operated, these suites are managed and configured from the ground and sequentially commanded through each maneuver of the RPO sequence. It is expected that significant parts of the system management can be migrated onboard by leveraging Commander, as discussed in our next example mission.

To demonstrate an example RPO scenario for a pair of vehicles in LEO, shown in Figure 10, Commander subscribes to telemetry from Navigator and Autopilot. In this mission, Commander steps through the process of

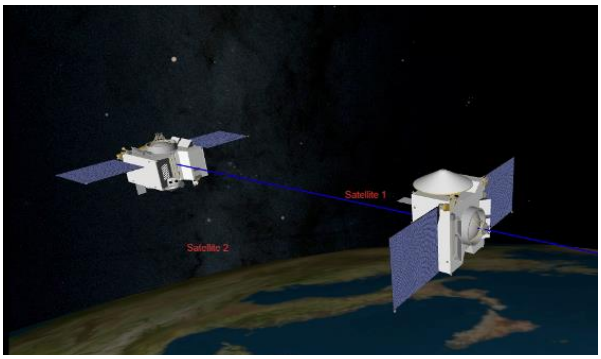

Figure 10: Visualization of RPO Mission in Progress

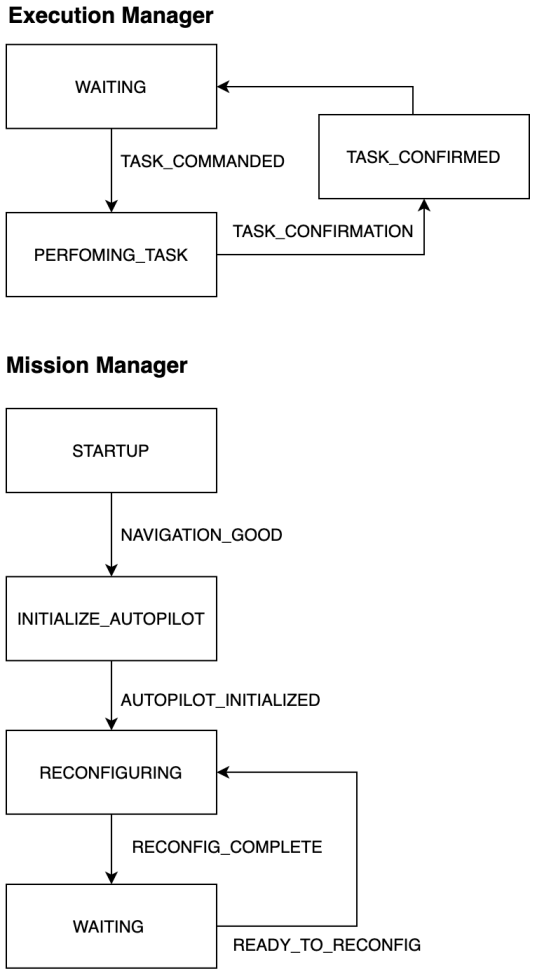

Figure 11: Simplified RPO State Diagram Summary

initializing Autopilot for autonomous maneuvering. Simplified system state diagrams are shown in Figure 11 and are now described. In this mission, MM is used primarily to interact with Navigator and Autopilot, and EM is used to manage TM After MM detects that Navigator has a navigation solution, it initializes Autopilot for the target spacecraft and operating mode. Once this initialization step is complete, Autopilot is ready to receive reconfiguration command(s) commanding changes in the reference or relative orbit. Commander then monitors navigation data until the vehicle reaches a preplanned orbital state. Once that state is reached, Commander sends a reconfigure command to Autopilot. By chaining together several successive reconfiguration and conditional waiting states, Commander can manage Autopilot through the maneuver phases of the RPO. This allows operations to be managed onboard instead of relying on the ground to advance the system to the next phase. This configuration allows Commander to advance semi-autonomous FSW

capabilities toward fully autonomous operations reducing operator workload and enabling new mission CONOPS.

Constellation Intelligence, Surveillance, and Reconnaissance

Commander also has the capability to manage large-scale distributed missions such as tasking constellation for an ISR constellation. This mission is applicable to upcoming deployments proposed by NASA, MDA and the SDA. A simple demonstration of the concept can be achieved by combining an external planning process with a simple onboard state machine using Commander. In this mission, the external planner can be either a human operator or a ground optimization process; in either case, it is responsible for assigning zero or more targets to each friendly asset. This assignment is represented as a simple Commander state diagram, such as Figure 12. Each spacecraft detects targets as they become visible, and send some appropriate tracking commands. For example, the spacecraft can slew to center the target in its field of view and capture data with a high-resolution camera.

Emergent has taken initial steps to integrate Commander with Cirrus to enable onboard target detection. We plan to extend this concept to demonstrate a simple ISR mission with autonomous responses to processed data, as shown in Figure 13. In this example, the targets are ships in the field of view of a sensor. An example of synthetic imagery with a ship target generated by Ascent being processed by a Machine Learning detection algorithm is shown in Figure 14. When a tasked observation results in a detection, Commander could initiate follow-up from other members of the constellation, it could queue additional observations from an alternate resource, or it could alert the ground of the detection.

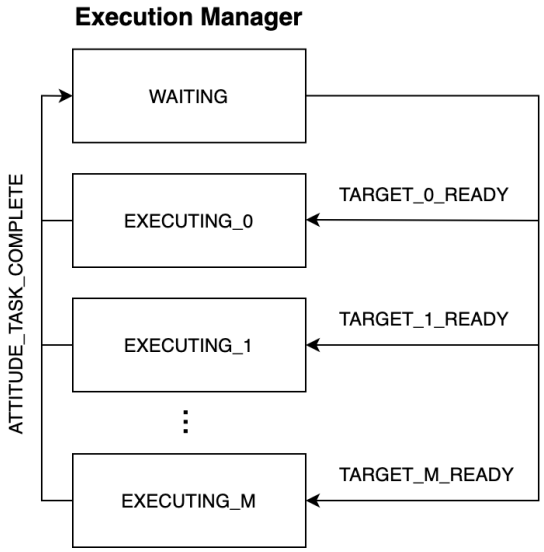

Figure 12: Simple ISR Execution Manager State Diagram

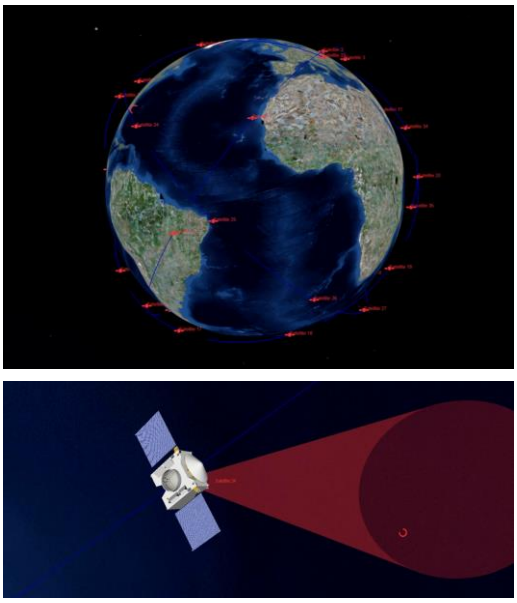

Figure 13: Visualization of ISR Mission in Progress

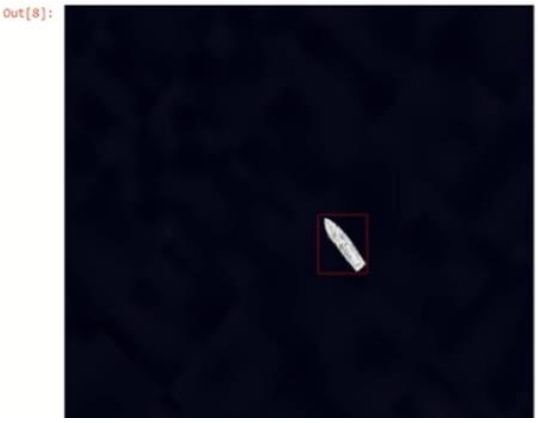

Figure 14: Example of ML Based Ship Detection from Ascent's synthetic data pipeline

CONCLUSIONS

The software applications discussed in this paper enable specific, subsystem-level capabilities on one or more spacecraft. The capabilities are general, but tailored for cooperating groups of smallsats. Navigator fuses sensor measurements within a cluster to enable absolute and relative navigation with robustness to hardware failures. Autopilot performs planning and execution of cooperative cluster tasks, such as maneuver planning and collision monitoring. Guardian performs configurable

FDIR using system and cluster-level health monitoring. Cirrus provides flexible cloud computing capabilities that enable compute and storage tasks to be performed on space platforms. These app suites can be executed individually, but feature various integrations that enable additional functionality.

The Commander flight software suite enables platforms to operate independently of ground commands for extended durations by monitoring telemetry from other apps and responding appropriately to events. Commander uses a software hierarchy with Mission Manager monitoring and coordinating activities across managed spacecraft, and Execution and Timeline Managers on each platform monitoring and coordinating onboard subsystems. Commander is a plan execution software suite, and the Mission and Execution Manager software utilizes a common framework for executing finite state machines.

By using Commander in conjunction with the other application suites, extended autonomous operation of smallsats is enabled. We demonstrate this in a discussion of representative missions: first, a simple autonomous lunar injection; second, an RPO mission using Navigator and Autopilot; third, a discussion of an ISR mission leveraging Cirrus for target detection. Cooperative space missions have achieved commercial success in low Earth orbit. For missions at higher altitudes, traditional operator-in-the-loop paradigms are costly and, in some cases, entirely impractical due to communications restrictions. By collectively enabling autonomous operation of smallsats, the software apps described in this paper can reduce costs for mission operators and enable the next generation of cooperative space missions.

References

1. Schmidt, J. & Phillips, M. A distributed, redundant navigation and fault detection system for DARPA system F6. in *AIAA Guidance, Navigation, and Control (GNC) Conference* (AIAA, 2013). doi:10.2514/6.2013-4963.

2. Chee, S. A. & Forbes, J. R. Norm-constrained consider Kalman filtering. *J. Guid. Control. Dyn.* **37**, 2048–2052 (2014).

3. O'Connor, B., Brown, A., Gordon, K., Schmidt, J. & Torre, R. De. A Service-based Architecture for Automated Guidance, Navigation, & Control Flight Software. *Work. Spacecr. Flight Softw.* (2013).

4. Brown, A. G. *et al.* Simulated Annealing Maneuver Planner for Cluster Flight. *Int. Symp. Sp. Flight Dyn.* 1–31 (2014).

5. Ruschmann, M. C. & McGreevy, J. Separable Architecture for Fault Isolation and Recovery. *31st Annu. AIAA/USU Conf. Small Satell.* (2017).

6. Brown, R. G. RAIM.pdf. in *The Global Positioning System* (eds. Parkinson, B. W. & James L. Spiker, J.) 143--165 (American Institute of Aeronautics and Astronautics, 1996).

7. Wilson, E., Sutter, D. W. & Mah, R. W. Motion-based thruster fault detection and isolation. *Collect. Tech. Pap. - InfoTech Aerosp. Adv. Contemp. Aerosp. Technol. Their Integr.* **4**, 2483–2517 (2005).