

Complete and Total Satellite Autonomy for SDA Applications, an In-Space Experiment

Dick Stottler and Evan Finnigan

Stottler Henke Associates, Inc.

Dick Wilkinson

Proof Labs

ABSTRACT

This paper will describe the results of an in-space experiment for a complete onboard autonomous satellite software system that performs every function of an SDA-oriented space asset, including self-preservation and mission planning/replanning, autonomously.

The onboard framework is called Total Autonomy and it controls all aspects of spacecraft operations including mission planning and scheduling and rescheduling in real-time based on sensed SDA threats, internal faults, or cyber attack; discerning SDA ground and space-based threats; safing and mitigation for these pop-up threats while rescheduling to safely maintain mission objectives to the degree possible; onboard subsystem (ADCS) fault detection, diagnosis, safing, mitigation, and rescheduling as required; and cyber attack detection, diagnosis, safing, and mitigation.

The Total Autonomy software was uploaded to the satellite and commissioned in July of 2026 and then updated and uploaded in August. For the experiment the satellite's primary mission is ISR, including discerning threats to space-based assets, including itself. This is an experiment, not a real ISR mission, with different objects playing the role of ground-based threats. Additionally, to gather a good volume of interesting data, faults and threats can be artificially inserted into the data streams. During commissioning, Total Autonomy did autonomously detect an uninjected fault in the ADCS. It was minor, but it was real and was appropriately flagged, logged, and reacted to.

The Total Autonomy framework includes the following modules, which execute independently and onboard:

Autonomous ISR Scheduling/Rescheduling

Autonomous Execution (including autonomous Safing)

Autonomous Execution Oversight

Autonomous Threat Detection/Reaction(Safing)/Recovery/Mitigation of Mission Goals

Autonomous In-Space Image Processing

Autonomous Fault Detection/Diagnosis/Reaction(Safing)/Recovery/Mitigation to maximally recover mission objectives

Autonomous Cyber Attack/Diagnosis/Mitigation

Imager Commanding

Experiment Fault/Threat Injection

These modules primarily interact through the Event Queue. The Autonomous Execution module pops the next event then calls the appropriate module to execute the event. The Autonomous Execution Oversight monitors the events being executed, especially their parameters, number and frequency. It is a small simple piece of code to double check to stop anything dangerous (e.g. pointing the imager too close to the sun) or to prevent anything nonsensical (e.g. way too many threats being autonomously discovered). Its job is to save the space hardware or the space experiment from a mistake in the Total Autonomy modules, since this is the first time they have all flown and they are all executing everything completely autonomously

Autonomous ISR Scheduling/Rescheduling module receives a list of on the ground target locations and time windows and optimally sequences them to capture the most requested images possible. It must also consider limitations imposed by ground-to-space threats (i.e. to not point the imager too close to a threat which may blind it) or imposed by faults in the components of the ADCS. Slewing time between images must also be taken into account. It places the most up-to-date schedule on the Event Queue as a series of ADCS commands to point at the desired locations and Imager commands to take one or several images (in rapid succession). New threats or new faults prompt it to remove the old scheduled events, reschedule, and populate the Event Queue with the new events.

The Autonomous Fault Detection/Diagnosis/Reaction(Safing)/Recovery/Mitigation module monitors the commands to the ADCS and the resulting estimated attitude parameters over time to detect and diagnose anomalies and, when appropriate, can itself cancel queued events and command safing actions. It then performs mitigating actions as required, including providing the ISR Scheduling/Rescheduling module with additional constraints

Similarly, the Autonomous Cyber Attack/Diagnosis/Mitigation module is monitoring spacecraft parameters to detect and diagnose cyberattacks on the vehicle and can take similar safing and commanding actions required to defeat them

When each image or set of images are taking, they are processed by the onboard Autonomous In-Space Image Processing module which determines if a threat exists. If so, this may prompt immediate additional imaging requests as well as immediate safing commands as needed to protect the imager. It will also provide location and/or orbit parameter information and related angle constraints to the ISR Scheduling/Rescheduling to allow it to avoid point the imager too close to the discovered threat.

Based on open literature review, we believe that this is the first entirely autonomous satellite system placed in space, given total autonomous control of its satellite and including all of the modules listed and described. The performance of each of these modules in their in-space environment and other results will be presented in the full paper.

1. INTRODUCTION/MOTIVATION

In addition to executing normal, planned mission tasks, total in-space, onboard autonomy requires the ability to quickly react to internal and external changes. Internal changes may be component faults or additional tasks generated by executing previously planned tasks. External changes could be environmental, such as space weather, or adversarial, either their actions or detecting their presence. When these changes occur, ground control may not be possible, either because there is no visibility with a currently functioning and available antenna on the ground, or communication equipment onboard is internally or externally (e.g. due to adversarial actions) disabled. Even if communication is immediately possible, an onboard reaction capability will always be quicker, since there are no communication and transmission delays. Also, typically there is more information available onboard than is always transmitted to the ground and in some important scenarios, reaction speed matters.

2. PROCESS OF GETTING INTO SPACE

2.1 Persistence Description

Total Autonomy is running on Mission Persistence run by Mission Control in Ottawa, Canada. It is an in-orbit ML testbed hosted on the Lemur-2 Krish satellite, built and operated by Spire Global. It is a 6U CubeSat orbiting in a sun-synchronous, low-earth orbit with an altitude of ~597 km. The primary hardware payloads on the satellite are an imager and a Marsupial computer. The Marsupial has a Xilinx Zynq Ultrascale+ MPSoC processor (model ZU4CG-1E) with a 1.3 GHz clock speed and two processing cores. The satellite's ADCS sensors include integrated 3-Axis Magnetometers, coarse and fine Sun Sensors, Earth-horizon sensors, Inertial Measurement Units (IMUs), and GNSS Receiver. The ADCS's actuators are a three-axis reaction wheel assembly for fine rotational maneuvers and momentum management, paired with electromagnetic magnetorquers (torque rods) for desaturation and magnetic-based control. For the Total Autonomy experiment, given the payload was an imager, we chose our "primary mission" to be earth observation. While the satellite is owned and operated by Spire Global, the ML testbed including the camera and Marsupial computer that comprise the payload are owned and operated by Mission Control out of their HQ in Ottawa. The Total Autonomy experiment is run in coordination with Mission Control who manage the payload schedule and tasking. For more details on the platform, architecture, testing setup and processes, and Mission Control's own flight experiments, we refer you to [1].

2.2 Fielding Software in Space

There are a number of limitations and challenges inherent with running software on a satellite in space. First, once uplinked to the spacecraft, the software cannot be easily modified. Our code delivery and uplink process takes two

weeks from code delivery to the satellite operator to commissioning in space. Additionally, processors on satellites tend to be low-performance compared to desktop processors, which means that runtime efficiency is very important. Finally, upload software package size will be limited. Based on the above, many of the algorithms for the autonomous modules were simplified versions. For example, for the image processing to locate predefined threats, we used scale-invariant feature transform (SIFT).

Setting up a developer test environment that is similar to the real satellite (referred to as the flight model) while still offering easy testing is very important for enabling code that works in test to also work in space. Our developer test environment for Total Autonomy consisted of a fully software-based test environment built by Mission Control called SANDBOCS, and a development board similar to the flight computer. SANDBOCS has mock satellite endpoints that enable testing calls to the real satellite API in a software only environment. Testing at this level primarily verifies the code logic. However, we did learn that testing code in a software environment on a modern computer can hide programming errors such as race conditions that can then manifest on an embedded processor later. In our case, this was because the high-performance computer was capable of opening, processing, and releasing files very quickly which reduced the risk that multiple threads would try to access the same file at the same time.

Testing on the development board, enables both performance testing and testing for more complex logical issues like race conditions. In our case, the image processing was the primary computationally intensive operation, and testing on the development board enables us to get a rough idea of how long it would take. This was important for determining image size for processing as well as how many images could be processed in a single payload window.

Throughout the process of testing and deploying Total Autonomy, we did learn that it is important to not overly rely on the test environment, and to actively consider the real flight model environment. For example, our development test environments (both our development board and the software based test environment) did not include imaging delays that are present on the actual satellite. After the imager is triggered, it takes approximately 3 minutes for the images to be captured, and returned back to the application code. This significantly limits the number of images that can be captured in a single payload.

2.3 Process for In-Space Deployment

There is a multi-step process for in-space deployment of mission code. The first step of the process is testing on a development machine in a fully software-based test environment. At this stage, logical software issues and other bugs should be detected and fixed. After software-based testing, there is testing on a representative development board that we have on premises. The development board uses a similar kind of processor to the flight hardware, but is not exactly the same. The development board actually had 4 cores while we only have access to the two processing cores on the Ultrascale+. We increase the accuracy of this step of testing by limiting the software to a single core. This stage of testing helps us determine performance, and catch any bugs or issues that might arise on the embedded system.

After testing on the development board, the code is packaged and sent to the satellite integrator for testing on the payload development kit (PDK). The PDK is a higher fidelity hardware test environment that is more similar to the actual in-space hardware than the development board. PDK testing is largely similar to testing on the development board, and in our tests it did not produce different results than development board testing.

After PDK testing comes the highest fidelity level of testing before deployment on the actual flight hardware. This testing is performed on a high-fidelity flatsat, a ground-based hardware version of the satellite that has the same processors, imager, and communication delays as the in-space satellite. At this point, the software is also subjected to a security review. . Flatsat-based testing confirms that the software can operate as expected with the same imaging delays and other hardware based restrictions onboard the actual satellite.

Following Flatsat testing is deployment to the in-space satellite. The software is uplinked and installed on the flight hardware. After installation, comes commissioning where the software's functionality is verified in space. In commissioning, the software is run on the satellite with simple parameters to verify that its basic functionality is correct. Once that functionality is verified, space operations begin.

3. HIGH LEVEL ARCHITECTURE DESCRIPTION

Our central premise that we set out to prove is that total onboard autonomy can be achieved by an onboard architecture consisting of a high-speed scheduling and reactive rescheduling capability, an event/action queue which consists of the schedule of actions to be executed, an execution module with associated execution monitor, and the various necessary independently executing processes such as fault and threat detection.

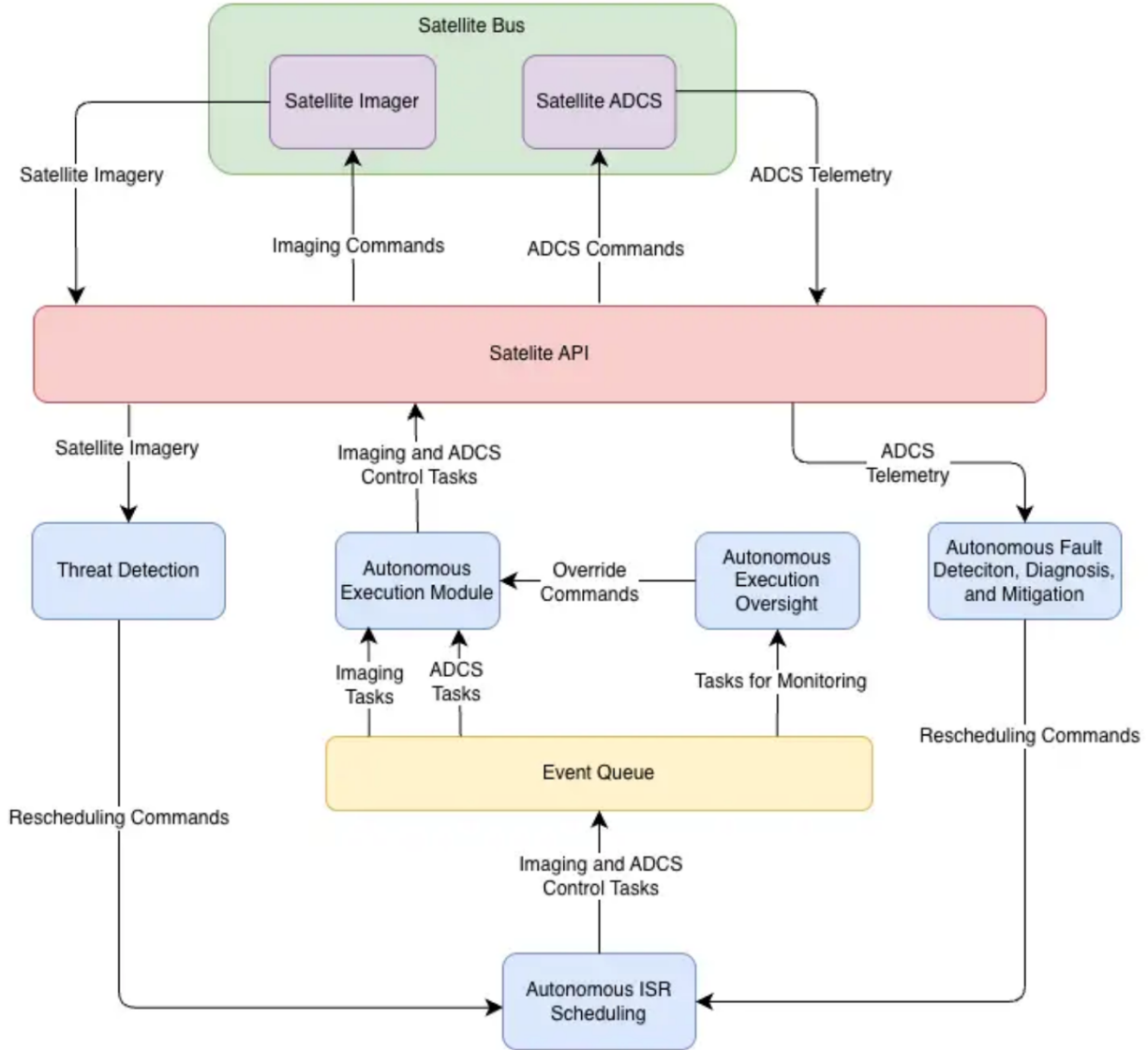


Fig. 1. High-Level Total Autonomy Architecture

The genesis of this concept was proposed in [2] where fault management (i.e. properly responding to system faults) can be viewed as a symbolic closed loop controller similar to the closed loop shown above. Fault management typically refers to internal faults (or changes to use the nomenclature from above). Our contribution was simply to extend this same notion to external changes, and externally- or internally-generated additional tasking. Autonomy requires quickly reacting to these external and internal changes at the appropriate time scales. For example, sudden faults or very immediate threats must be responded to quickly while slow degradation of components or non-urgent threats can be responded-to at longer time scales

4. HIGH-LEVEL DESCRIPTION OF COMPONENTS

4.1 Event Queue

The Event Queue (i.e. List of scheduled activities to execute) is just the simple, time-ordered list of time-stamped events to be executed. All modules can examine the queue and request insertion of new events, though the latter action prompts the reschedule module to determine if the new event requires that the events should be rescheduled. For example, a new imaging request may require a rescheduling of some of the other imaging events. Each event has the appropriate attributes to allow it to be properly executed and to allow appropriate rescheduling (e.g. the necessary resources, constraints, and time required to perform (or a method to calculate this) must be included). With the exception of rapid safing reactions, all actions to be executed first appear on the event queue to be popped off, one at a time, to be executed at the appropriate time, based on each's time-to-be-executed field. The types of events on the event queue include:

- Uploading (requested new tasking, config files, etc.)
- Scheduling of planned events
- Downloading (Logs, Results of mission tasks, threat and fault reports)
- Capturing images (i.e. Mission task to be executed)
- Looking for Threats (for us, just capturing an image)
- Processing a possible threat image](i.e. recognizing if a threat is present in the image)
- Mission tasks such as process images (simple acceptable image or not, e.g. processing for bad weather)
- Reactive Rescheduling
- Could also include initiating contact with the ground because of an unexpected event
- Mitigation Tasks, if not immediate rapid safing reactions (added by Fault Detection and Diagnosis)

The Event Queue keeps the list of events/actions sorted chronologically primarily includes methods to Pop the next Event, Insert and Delete Events, with changes being checked to determine if Rescheduling should be called.

4.2 Autonomous Execution

The Autonomous Execution module pops the next event, waits until the time-to-begin-execution, then calls the appropriate module to execute the event. For the most part, these calls are obvious. "Scheduling" and "Rescheduling" are synonyms and executed by one body of code, which takes as input the old schedule (i.e. the event queue, along with additional requested activities (e.g. additional requested images) and/or additional constraints (e.g. avoid pointing near a specific threat) and reoptimizes the schedule. For any of the image events, the imager is called with the exact time to begin imaging and the number of images to take over what (short) duration of time. A complication is that the imager had an unspecified delay from beginning to execute to when the first image was captured. Given that the satellite is orbiting at 7500 meters/second, this was non-trivial, but was worked out with some trial and error. Processing a possible threat image resulted in a call to the computer vision module to determine if the threat was actually present. One additional special case was injecting synthetic data into the telemetry stream. This usually related to mimicking different kinds of ADCS faults and cyber intrusions, since none of these types of situations were expected to actually occur and to show that the software reacted appropriately.

4.3 Autonomous Execution Oversight

As shown above, Autonomous Execution is paired with Autonomous Execution Oversight which monitors the events being executed, especially their number and frequency. Several past failures of spacecraft were due to repetitive active actions taken by onboard systems. Commonsense would have allowed a human to quickly stop these series of actions, put the spacecraft in a safe state, and determine what the root problem was (often a malfunctioning sensor). But, of course, no human is onboard or routinely available to an autonomous satellite. So this commonsense activity must be mimicked in software. Basically, it stops anything nonsensical that is going on (e.g. way too many threats being autonomously discovered.) Its job is to save the mission and it overrides execution based on monitoring the executed activities, especially their number and frequency and determining if any action violates a safety constraint.

4.4 Autonomous Rescheduling

Autonomous Rescheduling is called when changes to the events warrant it, when a new threat is discovered (which will cause safing actions and introduce additional constraints (such as keeping the imager from pointing too angularly close to any threat) and when a fault is detected that impacts the schedule, either due to mitigating actions, recovery time, or decreased capability. Autonomous Rescheduling must utilize tactical scheduling algorithms, that is, those that produce good results very quickly. The most common are priority-ordered greedy algorithms, but these should be avoided because although they execute very fast, they produce very poor results. The two main decisions that a tactical scheduler must continually make is which next task to consider, and which resources for which time periods should be assigned to that task. A good heuristic for the former is to schedule the hardest-to-schedule, most-constrained tasks first, since putting these off in the process is going to make it more likely that they cannot be scheduled at all. For the latter, a good tactical scheduler will not just consider the task, performance time window, and specific resources to be assigned, in isolation. For each task, for each assignment, it should consider how all other tasks and resources will be impacted. E.g. don't assign a resource and time window to a task, if there are other options and the first option would preclude being able to schedule another task (because it really needs those resources at that time). Of course, a good tactical scheduler needs a clever scheme to be able to very efficiently consider all other tasks and all other resources when making each individual resource assignment decision.

4.5 Autonomously Looking for Threats

Autonomously looking for threats involves scanning and searching with threat-specific sensors and then processing the resulting sensor data to detect (then track, if appropriate) threats to the spacecraft, its payloads or its mission. For our satellite, our only relevant sensor was an imager, so this module captures an image at the appropriate time and imager pointing angles then calls the Processing a Possible Threat Image Module.

4.6 Processing a Possible Threat Image

This module performs autonomous in-space image processing via the Scale-Invariant Feature Transform (SIFT) which is an algorithm for detecting distinctive points in an image and describing their appearance. The SIFT algorithm generates a 128-dimensional descriptor for each distinctive point that describes the image gradients near that point. This feature descriptor is invariant to scale and orientation, so the same physical feature in an image can be recognized even after it is enlarged, reduced, rotated or the camera viewpoint changes. This module uses SIFT features to match SDA threat image patches to images taken by the satellite imager in-flight. Our SIFT based matching algorithm finds distinctive points in the threat patch and the image obtained by the imager and detects a match between the two if they have the same set of SIFT features.

4.7 Autonomous Fault Detection/Diagnosis

Fault Detection includes the two independent technologies for fault detection: MBR and SOMs, a highly effective fault detection ML technique. The Fault Detection module receives fault detection notifications from each of the two technologies, over time, and executes a Confirmation/Reconciliation procedure which considers each input (across both the two technologies and across different periods of time) and forwards the combined result to a Diagnosis module, which uses the given information and MBR to identify the specific faulty component and likely root causes, if possible, or a candidate set of culprits, otherwise. Diagnosis also considers any outputs from Cyber Attack Detection.

These MBR and ML techniques have been previously applied for fault detection and diagnosis for several subsystems, including NASA's Astrobotic's Vertical Solar Array Technology for Lunar Traverse (VOLT) lunar rover, selected by Astrobotic for lunar operations, the ISS Experiment EPS/SEU-Recovery, Coriolis satellite, ISS's Urine Processing System, NASA Ames graywater system, lunar Gateway EPS HW model, the next generation xPLSS space suit, the Mars transit vehicle, Montana State University's LabSat, and Johnson Space Center's Advanced Life Support System.

4.7.1 Model-Based Reasoning (MBR)

One approach to detection, diagnosis, and characterization is Model-Based Reasoning (MBR). Model-based detection and diagnosis systems encode detailed and explicit descriptions of the interrelated factors that affect

subsystem behavior. These models typically represent the world as a collection of components, where each component is characterized by attribute values and one or more modes. Constraints specify required relationships among attribute values and modes, and constraint violations are used to identify components in faulty modes. For example, current, voltage drop across a resistor, and resistance are constrained by an algebraic relationship. If the sensed voltage, current, and known resistance do not obey this relationship, then either one of the sensors or the resistor might be at fault. Model-based reasoning systems have traditionally been used to diagnose faults in engineered systems. System components are characterized by nominal and faulty modes, and constraints on interconnected components are based on physical laws and design intent. Because models encode the effects of contextual factors, they can be applied reliably across contexts, such as the current environment, configuration, and sent commands. Model-based reasoning requires knowledge engineering efforts to encode these interacting effects. MBR engines can be extremely fast and do not necessarily require a large amount of memory or compute power, even for complex models.

During normal operations, the model is used to simulate the current behavior and compare the simulated sensor output values to the actual sensor outputs. Significant deviations are used to detect some kind of fault. The model is then used to reason which component faults are most likely to lead to the currently deviating sensor values. The set of possible faults (including sensor faults) which explain the sensor values is the MBR diagnosis engine's output. The process of using the model to diagnose failures is considered somewhat analogous to the reasoning an engineer uses when using a schematic to try to diagnose the fault.

4.7.2 Self-Organizing Maps (SOMS)

Our Machine Learning approach to fault detection uses a combination of SOMs (Self-Organizing Maps) and Case-Based Reasoning to detect known and unknown faults, and it is extremely accurate at doing so. Because data is generally not (as) available for fault scenarios, it is expected that SOMs are generally trained on nominal (i.e., non-faulty) data. Then, given a new data point, the SOM will classify the point as nominal or anomalous. However, anomalous does not necessarily mean faulty. There is the obvious case of false positives, but there is an even deeper issue. Anomalous only means "different from the training data" – so if the SOM was not trained on data that reflected a similar enough operational setting to the current data sample, the SOM would classify the current data sample as anomalous. However, if the SOM was trained on data from all possible nominal scenarios, then an anomaly is likely a fault. In this case, the SOM will not indicate which type of fault, but merely that it is an (unknown) fault. Note that in this situation, the SOM detects but does not diagnose a fault.

On the other hand, sometimes enough data is available for one or more (but not all) fault scenarios. In this case, multiple SOMs will be trained – one SOM for the nominal case, and one SOM for each faulty case. For example, if we had training data for nominal data and two types of faults, we would train 3 SOMs total. For simplicity, we will call them nominal-SOM, faultType1-SOM, and faultType2-SOM. Now, each SOM is trained to classify "normal vs abnormal" for its case but what "normal" and "abnormal" mean for each SOM is different – "normal" refers to what the SOM was trained on and "abnormal" refers to all else. Given a new data sample, nominal-SOM will classify it as nominal or not, faultType1-SOM will classify it as faultType1 or not, and faultType2-SOM will classify it as faultType2 or not. When a new data sample arrives, it is evaluated by each SOM, and one SOM is deemed to be the "match". If the match is nominal-SOM, then the data is deemed nominal. If the match is faultType1-SOM, then the data is deemed to be of faultType1. If the match is faultType2-SOM, then the data is deemed to be of faultType2.

The "match" is determined using Case-Based Reasoning (CBR). CBR is an AI methodology that solves problems by retrieving solutions to previous similar problems and altering them to meet current needs. In this context, there is a nominal case, and one or more faulty cases. The core attribute of a case is a SOM. A case will include other pertinent information such as an explanation of the operational mode and, where necessary, a recommended set of actions to address or mitigate the situation. SOMs serve as the index into the case-bases for retrieving cases resembling the situation of interest. We use the Minimum Quantization Error (MQE) as the similarity metric for this purpose. CBR will first use the MQE measure to find the closest reference case. It will then compare the MQE from the target data to predefined upper and lower thresholds of the training data MQE for the reference case to test whether the two situations are similar. If the target MQEs fall outside the threshold range, this is a sign that there is a significant deviation between the two. This implies that no case can be found that matches the MQE for this target case (i.e., we have encountered a novel anomaly). On the other hand, if the target MQEs fall within the threshold

range of the reference SOM's MQE, then the target and reference situations are similar enough for effective knowledge transfer. (To avoid confusion, we note that this CBR matching process is similar but different and distinct from the process that occur within a SOM which finds the closest matching SOM output node. SOM output node matching will be described in the SOM description section.) Note that for each SOM, the threshold can be adjusted (again, based primarily on the benefit/cost of True Positives, True Negatives, False Positives, and False Negatives). An improvement of this scheme will instead group data into batches of size N (an adjustable parameter), and have every SOM evaluate every sample in the batch. As mentioned before, this is to combat random noise/spikes. In this context, the number of consecutive abnormalities found, the fraction of the batch deemed anomalous, and/or an average MQE of the abnormal samples within a batch can be used instead of the MQE for one sample to find the match. Note that in this application of SOMs, where sufficient data is available for one or more fault types, the SOMs in combination with CBR do **detection** and **potential diagnosis** (depending on whether the SOMs+CBR think the supposedly faulty situation is a known fault or an unknown fault).

Self-Organizing Map Background

ADTM leverages the benefits of a Self-Organizing Map (SOM) for implicit data clustering and anomaly detection. Also known as a Kohonen map, a SOM is a two-layer artificial neural network (ANN) that uses unsupervised learning to produce a low-dimensional representation of the training samples. Inspired by the way sensory input (auditory, olfactory, tactile, etc.) map to specific areas of the cerebral cortex, SOMs are also tuned to various patterns of input data during training (Fig. 2). Consistent with this analogy, the nodes in the output layer of a SOM are also called “neurons.”

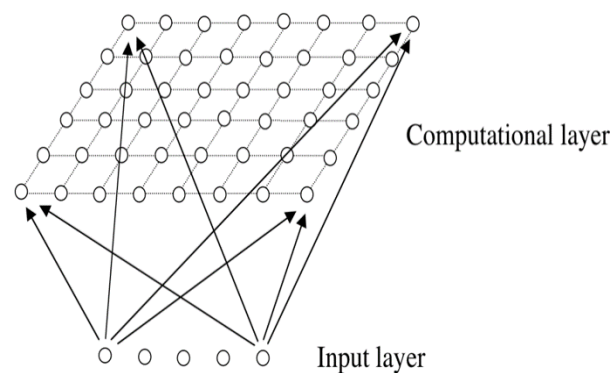


Fig. 2: Self-Organizing Map Diagram

The goal of training a SOM is to transform incoming inputs to a 1- or 2-dimensional map in a topologically ordered fashion such that points that are close together in the higher-dimensional input space are close together in the lower-dimensional output space as well. This mapping allows us to detect patterns of normal or anomalous behavior in a system, as different types of behavior map to different output units.

Specifically, the N -dimensional input data is fed into the SOM in the first layer and fully connected to a lattice of output neurons in the second layer. Each output neuron is associated with a N -dimensional weight vector. We represent an output neuron by its two-dimensional coordinate of its position in the grid. Like the clustering technique k -means, the two dimensions of the grid are parameters that are tuned during model validation. Unlike k -means, however, the clusters learned during SOM training are topologically ordered through the following competitive learning process:

Each input vector is compared with the N -dimensional weight vector associated with each output node. The closest output node is chosen as the winner, or ‘Best Matching Unit’ (BMU), where ‘close’ is defined by a distance function (we chose Euclidean distance). Each BMU is associated with an entire neighborhood of related neurons whose weight vectors are also updated, though to a lesser extent, proportional to their distance to the BMU in the 2D output lattice.

In other words, entire neighborhoods of related neurons get updated in the direction of the input data that is closest to them, so that the topology of the N -dimensional input space is preserved in the 2-dimensional output space. We can think of this learning rule as pulling the weight vector associated with the BMU in the direction of the input vector. All neurons in the same 'neighborhood' are also dragged along, but to a lesser extent.

4.7.3 Hybridizing MBR and SOMS

Hybridizing MBR and SOMs emphasizes the benefits of each and minimizes the disadvantages. Generally, the benefits of MBR coincide with the disadvantages of SOMs, and vice versa, so the two complement each other well. In addition, because MBR and SOMs are very different technologies (especially in that one is model-based and one is model-free), their agreement provides extra confidence in the result (compared to using only MBR or only SOMs). A hybrid fault detection system uses both SOMs and MBR to monitor sensor values to identify anomalous behavior. When an anomaly is detected, there are a few possibilities: both systems detect it, only the SOMs detect it, or only the MBR system detects it. The advantages of a hybrid MBR-ML detection and diagnosis system are that it:

- Utilizes a priori design knowledge when it exists and it is efficient to do so (from MBR).
- Utilizes real or simulation sensor data when it exists and as it is produced (from ML).
- Utilizes very independently developed anomaly detection technologies for increased assurance.
- Can use its model to diagnose, reconfigure, recalculate, and replan in reaction to newly detected faults (whichever technology detected them) and can explain its reasoning (from MBR).
- Can discover unknown or unmodelled relationships between sensor data in different modes. These relationships can span across subsystems (from ML).
- Can detect and diagnose anomalies never before encountered or trained for, using its MBR component and handle rare (but modelled) operating conditions.
- Can use the ML component to disambiguate overlapping MBR states.

4.8 Autonomous Cyber Attack Detection

A special case within Fault Detection is Cyber Attack. Total Autonomy has an independent Cyber Attack Detection module which analyzes the ADCS telemetry and which serves two separate purposes. In cases where the Fault Detection module detects an anomaly in the telemetry, this module serves, if it finds evidence of an intrusion, to explain the root cause of that anomalous telemetry. Secondly, this module examines telemetry from a different perspective and may therefore find anomalies that the Fault Detection module does not, i.e. evidence of a cyber intrusion that does not violate any of the design parameters or normal operating telemetry of the ADCS. Because the Cyber Attack Detection module is based on different philosophies and scenarios from normal Fault Detection (which is primarily looking for faults of an internal engineering nature) it often detects anomalies that MBR or even ML does not. This was true the majority of the time in our experiments. Our Cyber Detection module was developed under the direction of Proof Labs. They have a very interesting and unique perspective in that their experience includes successful offensive satellite cyber operations. I.e. they have hacked into satellite systems and therefore can very realistically describe the type of evidence that should be looked for as well as realistic test cases. (And, of course, this unique perspective allows them to excel at defensive satellite cyber intrusion detection).

4.9 Experiment Fault/Threat Injection

Because we did not expect any real ADCS faults, in order to test the system, we needed to be able to inject faults from an uploaded injected faults file. Similarly, we were not sure, in advance, how well the computer vision techniques would work at finding threats so we included the ability to insert threats. These capabilities did give us the opportunity to test many different situations and combinations of situations. As a sidenote, we initially thought that this module was merely a side-effect of this project being an experiment and that in a real operational system, injecting faults and threats would be inappropriate. However, on further reflection, as long there was some way to disable any actions not appropriate in reaction to a fake fault or fake threat, it would be a good idea to include this ability even in an operational system in order to be able thoroughly test the system in space.

5. RESULTS

At this point, all components of Total Autonomy, described above, have successfully executed in place, including finding a real and unexpected ADCS anomaly. All fault and threat situations have been handled correctly, both singularly and when they have occurred during the same pass. The only real problems were the result of cloud cover over the area of an expected threat. I.e. the system could not find the threat if the threat was not discernible through the clouds that were present at the time that the images were captured. Specifically, the following situations have been autonomously successfully handled by the in-space processor, including corrective/mitigating actions and rapid rescheduling:

Table 1. Various Situations and Total Autonomy's Reaction to them

Situation	Total Autonomy's Reaction
Basic Real Time Test, Nominal Operations	Receive set of desired observations and optimally schedule them
Actual Unexpected ADCS fault occurs	And is Automatically detected
Threats checked for but not found	System correctly determines that a threat is NOT present
Single Threat Detection	Onboard computer vision processing finds threat in an image, new constraints checked, rescheduling is not necessary
Injected ADCS fault with subsequent observations	Injected fault detected, resolved and image tasks are rescheduled
Injected ADCS sensor and effector faults	Both are detected, resolved, and image tasks are rescheduled
Injected ADCS fault and Injected threat	A threat is injected after the first observation point which causes another observation point to be removed from the schedule because it is now threat adjacent. A fault is injected causing the last observation point to be rescheduled.
Injected ADCS fault and real threat	A real threat is detected in an image captured of the first observation point. This threat causes a later observation point to be removed from the schedule because it is now threat adjacent. The last observation point is rescheduled because of an injected ADCS fault.
Injected threat that affects schedule	A pre-existing injected threat makes it so one observation point cannot be executed and a swap will have to be performed.
Real detected threat affects the schedule	A newly found real threat in the first observation point causes an observation in a subsequent orbit to need to be removed from the schedule.
Excessive injected threats	Threat list considered suspect
Version 2 Uploaded	
Actual Unexpected ADCS fault occurs	Both MBR and SOM detect same fault (i.e. they agree)
5 different cyber attack scenarios	MBR detects none, SOM detects 1, Cyber Attack Detection detects all 5

To illustrate some of the above, extractions from the log files, downloaded from the satellite, are reproduced and explained below.

Nominal Operations

Every pass, starts the same as the first 7 lines below, starting up all the independent threads of all the modules and reading in the updated list of requested observations. The Scheduling thread schedules all of the imaging events and adds them to the event queue, one at a time (only the first one is shown). The last 5 lines show the imaging action being executed. All pass has similar elements so these will not be repeated in the other examples.

INFO - 1784460244.643299 - Starting threads
INFO - 1784460244.6440575 - AutonomousExecution thread started
INFO - 1784460244.6447887 - AutonomousExecutionOversight thread started
INFO - 1784460244.6452682 - Scheduling thread started at time 1784460244

```

INFO - 1784460244.6465387 - FaultDetection thread started
INFO - 1784460244.6478395 - ThreatDetection thread started
INFO - 1784460244.6471245 - File /home/user/uploads/sample_observations_test_11.json loaded
INFO - 1784460244.6483417 - Extracted new Image event at time 1784460244
...
INFO - 1784460245.609535 - MoveToEvent(id = 1101, lat_lon_target = (30.05, 31.24)) added to log file for time
1784460252
INFO - 1784460245.6105204 - ImageEvent(id = 1101, number_of_images=4, duration_to_take_images=4,
lat_lon_target = (30.05, 31.24)) added to log file for time 1784460255
...
INFO - 1784460255.026783 - Execute Image event at time 1784460255
INFO - 1784460396.3768985 - Image capture__timestamp_1784460255.0275512_id_1101_series_num_0 saved at
1784460255.0275512
INFO - 1784460396.3782442 - Image capture__timestamp_1784460255.0275512_id_1101_series_num_1 saved at
1784460255.0275512
INFO - 1784460396.3793328 - Image capture__timestamp_1784460255.0275512_id_1101_series_num_2 saved at
1784460255.0275512
INFO - 1784460396.4129205 - Image capture__timestamp_1784460255.0275512_id_1101_series_num_3 saved at
1784460255.0275512
...

```

Actual Unexpected ADCS fault occurs – And is Automatically detected

The unexpected ADCS seems to be occasionally tied to imaging events, so the relevant imaging event is shown being extracted from the requests file then executed (actually a series of 4 images). At time, 1784454690, the Fault Detection module detects the unexpected fault and 7 seconds later the fault clears.

```

...
INFO - 1784454492.961807 - ImageEvent(id = 1101, number_of_images=4, duration_to_take_images=4,
lat_lon_target = (30.05, 31.24)) added to log file for time 1784454502
...
INFO - 1784454502.0725737 - Execute Image event at time 1784454502
INFO - 1784454645.5042448 - Image capture__timestamp_1784454502.073337_id_1101_series_num_0 saved at
1784454502.073337
INFO - 1784454645.5054011 - Image capture__timestamp_1784454502.073337_id_1101_series_num_1 saved at
1784454502.073337
INFO - 1784454645.5063925 - Image capture__timestamp_1784454502.073337_id_1101_series_num_2 saved at
1784454502.073337
INFO - 1784454645.5398533 - Image capture__timestamp_1784454502.073337_id_1101_series_num_3 saved at
1784454502.073337
...
INFO - 1784454690.3226883 - ADCS Fault detected: control_error_angle_deg fault: {'fault_code':
'CONTROL_ERROR_ANGLE_DEG_FAULT', 'deviation': 0.02978270617632528, 'confidence': 'medium', ...
INFO - 1784454697.4053228 - ADCS Fault cleared: control_error_angle_deg fault
...

```

Injected ADCS sensor and effector faults – Both are detected, resolved, and image tasks are rescheduled

```

INFO - 1784748210.1738915 - ImageEvent(id = 101, number_of_images=3, duration_to_take_images=3,
lat_lon_target = (43.503171, -85.70081)) added to event queue and log file for time 1784748239
INFO - 1784748210.186272 - ImageEvent(id = 102, number_of_images=3, duration_to_take_images=3,
lat_lon_target = (36.123061, -87.918388)) added to event queue and log file for time 1784748359
...
INFO - 1784748239.0260146 - Execute Image event at time 1784748239
...
INFO - 1784748375.5802102 - Execute Image event at time 1784748359

```

```

INFO - 1784748422.3559258 - ADCS Fault detected: control_error_angle_deg fault: {'fault_code':
'CONTROL_ERROR_ANGLE_DEG_FAULT', 'deviation': 0.007343545570510757, 'confidence': 'medium',
'diagnosis': <ADCSFaultDiagnosis.SENSOR: 'Sensor'>, 'timestamp': 1784748422.3413117}
...
INFO - 1784748452.7309988 - ADCS Fault detected: control_error_angle_deg fault: {'fault_code':
'CONTROL_ERROR_ANGLE_DEG_FAULT', 'deviation': 3.7938265879026587, 'confidence': 'medium',
'diagnosis': <ADCSFaultDiagnosis.EFFECTOR: 'Effector'>, 'timestamp': 1784748452.7227187}
...
INFO - 1784748501.08571 - Fault detected at time 1784748422
INFO - 1784748501.086338 - Resetting ADCS Sensor
INFO - 1784748501.2922137 - Fault detected at time 1784748452
INFO - 1784748501.293116 - Resetting ADCS Effector
INFO - 1784748501.4055934 - Image event rescheduled to next orbit due to fault present
INFO - 1784748801.0875387 - ADCS Sensor Reset
INFO - 1784748801.2942882 - ADCS Effector Reset Complete
INFO - 1784748801.8946888 - ADCS Fault cleared: control_error_angle_deg fault

```

Real detected threat affects the schedule - A newly found real threat in the first observation point causes an observation in a subsequent orbit to need to be removed from the schedule.

Image Event 1604 was the imaging event that was originally scheduled to happen on August 10, and it gets discarded because it is near the threat in Croatia that triggered the rescheduling. The other two events that get scheduled are other locations that are not near the threat so they are added to the event queue. Note that in the last two lines, the events are being added substantially far into the future, i.e. a future orbit.

```

INFO - 1786279236.471234 - Running threat detection for image id: 1601, series number: 0
INFO - 1786279395.7842636 - Threat detected at image position: (234, 125), latitude and longitude: (43.50056 ,
15.990161), for image id: 1601, series number: 0
INFO - 1786280024.48279 - Running threat detection for image id: 1601, series number: 8
INFO - 1786280106.1834278 - Threat detected at image position: (235, 240), latitude and longitude: (43.50056 ,
15.990161), for image id: 1601, series number: 8
INFO - 1786280178.4916012 - Image Event 1604 discarded because it is threat adjacent.
INFO - 1786280178.5106623 - ImageEvent(id = 1605, number_of_images=10, duration_to_take_images=10,
lat_lon_target = (32.227112, 14.445361)) added to event queue and log file for time 1786365867
INFO - 1786280178.5268257 - ImageEvent(id = 1606, number_of_images=10, duration_to_take_images=10,
lat_lon_target = (18.865375, 11.317574)) added to event queue and log file for time 1786366091

```

Injected ADCS fault and real threat - A real threat is detected in an image captured of the first observation point. This threat causes a later observation point to be removed from the schedule because it is now threat adjacent. The last observation point is rescheduled because of an injected ADCS fault.

Note in the first lines, the same threat is found in two different images because the images overlap and in the second to last line, the image event is moved forward in time by about 25 hours, 90,000 seconds.

```

INFO - 1786707073.0845735 - Running threat detection for image id: 1801, series number: 5
INFO - 1786707110.9860282 - Threat detected at image position: (250, 105), latitude and longitude: (30.07266 ,
31.142468), for image id: 1801, series number: 5
INFO - 1786707110.9864435 - Running threat detection for image id: 1801, series number: 6
INFO - 1786707151.2383013 - Threat detected at image position: (232, 236), latitude and longitude: (30.07266 ,
31.142468), for image id: 1801, series number: 6
...
INFO - 1786707250.9970777 - Image Event 1803 discarded because it is threat adjacent.
...

```

INFO - 1786707672.4617946 - ADCS Fault detected: control_error_angle_deg fault: {'fault_code': 'CONTROL_ERROR_ANGLE_DEG_FAULT', 'deviation': 3.7938265879026587, 'confidence': 'medium', 'diagnosis': <ADCSFaultDiagnosis.EFFECTOR: 'Effector'>, 'timestamp': 1786707672.4590492}
INFO - 1786707672.5053358 - Fault detected at time 1786707672
INFO - 1786707672.5058696 - Resetting ADCS Effector
INFO - 1786707714.236871 - Image event rescheduled to next orbit due to fault present
INFO - 1786707714.2789664 - ImageEvent(id = 1802, number_of_images=10, duration_to_take_images=10, lat_lon_target = (-31.207925, 17.807029)) added to event queue and log file for time 1786794357
INFO - 1786707714.2798471 - 1 event added to the event queue

Cyber Attack, mismatched eclipse flag – Both the SOMs and Cyber Attack Detection module catch this

INFO - 1787179085.6854267 - Cyber attack detected: {'cyber_attack_code': 'ECLIPSE_FLAG_CYBER_ATTACK', 'detected': True, 'rationale': {'issue': 'Eclipse flag is set', 'eclipse_flag': 1}}
INFO - 1787179085.687334 - SOM detected an anomaly: {'fault_code': 'SOM_ANOMALY', 'mqe': 3.6084681030773194}

Actual Unexpected ADCS fault occurs - Both MBR and SOM detect same fault (i.e. they agree)

INFO - 1778627032.3072803 - SOM detected an anomaly: {'fault_code': 'SOM_ANOMALY', 'mqe': 3.47340269629373}
INFO - 1778627032.324307 - ADCS Fault detected: control_error_angle_deg fault: {'fault_code': 'CONTROL_ERROR_ANGLE_DEG_FAULT', 'deviation': 0.00300000000000000027, 'confidence': 'medium', 'diagnosis': <ADCSFaultDiagnosis.SENSOR: 'Sensor'>, 'timestamp': 1778627032.4330945}
INFO - 1778627034.3332686 - Fault detected at time 1778627032
INFO - 1778627034.3338442 - Resetting ADCS Sensor
INFO - 1778627034.3349051 - ADCS Sensor Reset
INFO - 1778627035.3245158 - ADCS Fault cleared: control_error_angle_deg fault
INFO - 1778627035.3258884 - SOM anomaly cleared

6. LESSONS LEARNED / CONCLUSIONS

First and foremost, our central premise was well-validated, that total onboard autonomy can be achieved by an onboard architecture consisting of a high-speed scheduling and reactive rescheduling capability, an event/action queue which consists of the schedule of actions to be executed, an execution module with associated execution monitor, and the various necessary independently executing processes such as fault and threat detection. And that the system could be fielded and function with the limited computational resources available onboard. Total Autonomy performed flawlessly in a range of situations and combinations of situations, including real and injected faults, “real” and injected threats, and a mix of both. The only issues, related to cloud cover, and, given our imager, this was insurmountable. We could have added cloud and/or smoke obscuration vision processing to automatically reschedule capturing images when the first attempt was unsuccessful.

There were a number of more minor lessons learned. First, adequate logging is very important. It is not possible to start a debugger and step through the code on the satellite, and the only information that will be gathered about the software’s in-space operation are the logs that are generated while it is running. Therefore, the logs must contain all relevant information that will be required to interpret the software’s behavior or debug issues that arise. For our Total Autonomy implementation, we included logging relevant to the observation scheduling process, operating the imager, fault and threat detections, and whenever files are read or written to onboard storage. One additional key piece of information that we learned we should have logged is the ADCS telemetry data from when the imager is triggered. This would be useful for helping us georeference the images that we gather.

Another important software best practice for fielding software in space is implementing the software to handle exceptions and other unforeseen circumstances well. Our Total Autonomy system only got results retrieved a few times a week. With limited opportunities, it was preferable that even if there was a failure or exception the software could recover and some results could still be salvaged. Beyond handling exceptions, defensive coding strategies in general are valuable. For example, handling missing configuration files gracefully without crashing, handling

missing folders, and isolating higher risk operations like interacting with the file system from lower risk operations like writing to an in memory queue could all enable a payload window to complete with some results as opposed to not providing any usable results.

The software package size for Total Autonomy was very limited. However, the bandwidth to upload the software to both the satellite and the flatsat was even more limited. If we had used the maximum amount of memory available to us, it would have taken a week to load the software to the flatsat and another whole week to upload it to the satellite. By using substantially less program memory than we were allocated we were able to greatly shorten these time scales.

To operate within the CPU constraints, all of the software had to execute very efficiently. The vast majority of our techniques tend to have modest CPU requirements already, but an exception was the computer vision associated with threat detection. The solution was twofold. First, we used SIFT, which is very efficient, and second, we did not process the entire image, just the smaller patch where the threat was most likely to be located.

7. ACKNOWLEDGEMENTS

Mission Control acknowledges the support of the Canadian Space Agency [SIMA03P2].

8. References

- [1] Candela, A., Davis, E., Branch, A., Russino, J., Chien, S., Bonham-Carter, B., et. al, (2026) Towards Onboard Multi-View Analysis of Satellite Imagery: Monocular 2D Motion Estimation and 3D Scene Reconstruction, *CVPR Workshop, 2026*.
- [2] Robinson, P. (2013). Fault Management (FM) as a controller. *AIAA Infotech@Aerospace Conference*, Boston, MA, 2013.
- [3] Stottler, R., Ramachandran, S., Healy, C., Singhal, A., & Finnigan, E. (2022). On-board, autonomous, hybrid spacecraft subsystem fault and anomaly detection, diagnosis, root cause determination, and recovery. In *Proceedings of Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS), 2022*.