A STATE MACHINE ARCHITECTURE FOR AEROSPACE VEHICLE FAULT PROTECTION

A Dissertation
Presented to
The Academic Faculty

by

Peter Zane Schulte

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Daniel Guggenheim School of Aerospace Engineering

Georgia Institute of Technology
August 2018

A STATE MACHINE ARCHITECTURE FOR AEROSPACE VEHICLE FAULT PROTECTION

Approved by:

Dr. David A. Spencer, Advisor
Daniel Guggenheim School of
Aerospace Engineering
Georgia Institute of Technology

Dr. E. Glenn Lightsey
Daniel Guggenheim School of
Aerospace Engineering
Georgia Institute of Technology

Dr. Mark Costello
Daniel Guggenheim School of
Aerospace Engineering
Georgia Institute of Technology

Dr. Neil Smith
Visual Computing Center
King Abdullah University of Science and Technology

Mr. Paul Rosendall
Embedded Applications Group
Space Department
Johns Hopkins University Applied Physics Laboratory

Date Approved: June 5, 2018

Soli Deo gloria

ACKNOWLEDGEMENTS

First and foremost, thanks to my wife Anita. I love you with all my heart, and I am beyond excited to walk beside you till our days are done. Personal thanks are due to many friends, family, and colleagues (too numerous to list here) who have supported me.

Thanks to my advisor Dr. David A. Spencer. He has been a wonderful mentor since recruiting me to Georgia Tech as a Master’s student to work on the Prox-1 student satellite project. His experience, insight, and advice have guided every step of this study, and he has frequently gone out of his way to ensure that my academic work, research, and career are successful and impactful.

Thanks also to Dr. Glenn Lightsey for his mentorship at The University of Texas at Austin and for continuing to support and encourage me after both of us moved to Georgia Tech. He has also provided a significant amount of administrative support to make my academic experience at Georgia Tech and this research study smooth and successful.

Paul Rosendall at the Johns Hopkins University Applied Physics Lab (APL) provided very helpful insights on the practical challenges and current state-of-the-art of fault protection. He was especially helpful in introducing me to the trade study APL performed to directly compare the rule-based and model-based fault protection paradigms.

FalconViz and the King Abdullah University of Science and Technology (KAUST) in Thuwal, Saudi Arabia provided funding support and technical guidance for development of the UAV Nervous System. Dr. Neil Smith provided ideas, mentorship, personal support, and technical guidance that were invaluable to me and to this study.

The Mars Sample Return fault protection work was completed under contract with the NASA Jet Propulsion Laboratory (JPL) at the California Institute of Technology. McClain Goggin at Purdue University completed most of the trajectory design work, developed much of the code for relative trajectory control and mode management, and originally wrote most of the descriptions of relative orbit elements and terminal rendezvous maneuvers that are included in this dissertation. Special thanks to Rob Lock (Mars Program Office) and Peter Meakin (Fault Protection and Autonomy group supervisor) at JPL for their mentorship and assistance. Also, the JPL Mars Sample Return study team provided many inputs by participating in breakout discussions:

Rendezvous team: Austin Nicholas, Alan Didion

- Navigation: Ed Riedel, Rob Haw, Eric Gustafson, Drew Vaughan, Dylan Boone
- Guidance/Control: George Chen, Jack Aldrich
- Sequencing: Chris Grasso, PJ Guske

Capture system team: Joe Parrish, Paulo Younse, Marco Dolci

Orbiter Flight System team: Kristina Larson, Ny Sou Okon, Travis Imken

Many engineers around JPL were consulted to solicit ideas for fault protection research, and their insights have been valuable: Lorraine Fesq, Kevin Barltrop, Jeff Levison, John Day, Bob Rasmussen, Mike Sievers, Magdy Bareh, Gene Lee, Garth Watney, Len Reder, Rob Bocchino, Tracy Neilson, Rajeev Joshi, Jean-Francois Castet, Ken Starr, Lermont Khachikyan, Bobak Ferdowsi, and John West.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1148903. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

$_0$	initial reference time (subscript)
3D	Three-dimensional
6DOF	Six Degree-of-Freedom
APL	Johns Hopkins Applied Physics Lab
$_{app}$	approach (subscript for v-bar hops)
a_r	relative semi-major axis (m)
ASCII	American Standard Code for Information Interchange
A_z	cross-track amplitude (m)
$_b$	Body-Fixed Frame (subscript)
DART	Demonstration of Autonomous Rendezvous Technology
EPS	Electrical Power Subsystem
E_r	relative eccentric anomaly (rad)
ESC	Electronic Speed Control
FCR	Fault Containment Region
FDIR	Fault Detection, Isolation, and Recovery
FMECA	Failure Modes, Effects, and Criticality Analysis
FOV	Field-of-view
FP	Fault Protection
FSW	Flight Software
FTA	Fault Tree Analysis
GN&C	Guidance, Navigation, and Control
GPS	Global Positioning System
HALO	Hydrology and Land Observation

I2C	Inter-Integrated Circuit
IAU	International Astronomical Union
IR	Infrared
JPL	Jet Propulsion Laboratory
KAUST	King Abdullah University of Science and Technology
KNN	K-nearest neighbors
LIDAR	Light Detection And Ranging
LVLH	Local Vertical Local Horizontal
MATLAB	Matrix Laboratory
MAV	Mars Ascent Vehicle
MSR	Mars Sample Return
MUBLCOM	Multiple Paths, Beyond-Line-of-Sight Communications satellite
n	mean motion of the target spacecraft (rad/s)
NASA	National Aeronautics and Space Administration
OS	Orbiting Sample canister
PITL	Processor-in-the-Loop
ProxOps	Proximity Operations
$_r$	relative (subscript for relative orbit elements)
$\hat{\boldsymbol{R}}$	Radial LVLH basis vector (x-axis)
r-bar	Forced motion approach to target via radial direction
$\boldsymbol{r}_{SRO}$	Position vector of the SRO in the Mars IAU frame
ROCS	Rendezvous Orbiting Sample Capture System
$\hat{\boldsymbol{S}}$	Along-track LVLH basis vector (y-axis)
S&MS	Structures and Mechanical Systems
SITL	Software-in-the-Loop

SPICE	Spacecraft, Planet, Instrument, Orientation, & Events ephemeris toolkit
SRO	Sample Return Orbiter
STEREO	Solar TErrestrial RElations Observatory
t	time (sec)
TBR	To Be Resolved
TT&C	Telemetry, Tracking, and Control
UAV	Unmanned Aerial Vehicle
UML	Unified Modeling Language
v-bar	Forced motion approach to target via along-track direction
$\boldsymbol{v}_{SRO}$	Velocity vector of the SRO in the Mars IAU frame
V&V	Verification and Validation
$\widehat{\boldsymbol{W}}$	Cross-track LVLH basis vector (z-axis)
^{x}	Skew function (superscript)
x	Cartesian radial relative position (m)
x_r	radial position of instantaneous center of motion (m)
$\dot{x}$	Cartesian radial relative velocity (m/s)
y	Cartesian along-track relative position (m)
y_r	along-track position of instantaneous center of motion (m)
$\dot{y}$	Cartesian along-track relative velocity (m/s)
z	Cartesian cross-track relative position (m)
$\dot{z}$	Cartesian cross-track relative velocity (m/s)
Δt	change in time $(\mathrm{t} - \mathrm{t}_0)$ (sec)
ΔV	change in velocity (delta-V)
ψ	cross-track phase angle (rad)

SUMMARY

Because of their complexity and the unforgiving environment in which they operate, aerospace vehicles are vulnerable to mission-critical failures. In order to prevent these failures, aerospace vehicles often employ Fault Detection, Isolation, and Recovery (FDIR) systems to sense, identify the source of, and recover from faults. Typically, aerospace systems use a rule-based paradigm for FDIR where telemetry values are monitored against specific logical statements such as static upper and lower limits. The model-based paradigm allows more complex decision logic to be used for FDIR. State machines are a particular tool for model-based FDIR that have been explored by industry but not yet widely adopted. This study develops a generic and modular state machine FDIR architecture that is portable to flight software. The study will focus on FDIR for the Guidance, Navigation, & Control subsystem, but it will be presented in a manner that is applicable to all vehicle subsystems. The state machine formulation is applied for on-board model-based fault diagnosis. Two specific case studies are employed to demonstrate the architecture. The first is a terrestrial application of unmanned aerial vehicles for 3D scanning and mapping, which is validated through flight testing. The second is a space-based application of automated close approach and capture for a Mars sample return mission, which is validated through software-in-the-loop testing with flight-like software components.

CHAPTER 1. INTRODUCTION: FAULT PROTECTION STATE-OF-THE-ART

1.1 Motivation

The capability to recover gracefully from hardware or software faults is critical for many aerospace applications. This is particularly true for autonomous missions involving proximity operations (ProxOps), where multiple vehicles are operating at close range. Previous ProxOps missions have experienced faults that resulted in a failure to meet mission objectives. For example, NASA’s Demonstration of Autonomous Rendezvous Technology (DART) spacecraft was designed to rendezvous with the Multiple Paths, Beyond-Line-of-Sight Communications (MUBLCOM) satellite in space and perform ProxOps maneuvers, as shown in Figure 1. DART experienced a mission failure when it collided with MUBLCOM during automated operations due to software errors that led to inaccurate range estimation. In a failure investigation report for DART, NASA recommended that “designers for such spacecraft should develop and adhere to a robust, detailed set of requirements for fault detection, isolation, and recovery in order to prevent a mishap” [1].

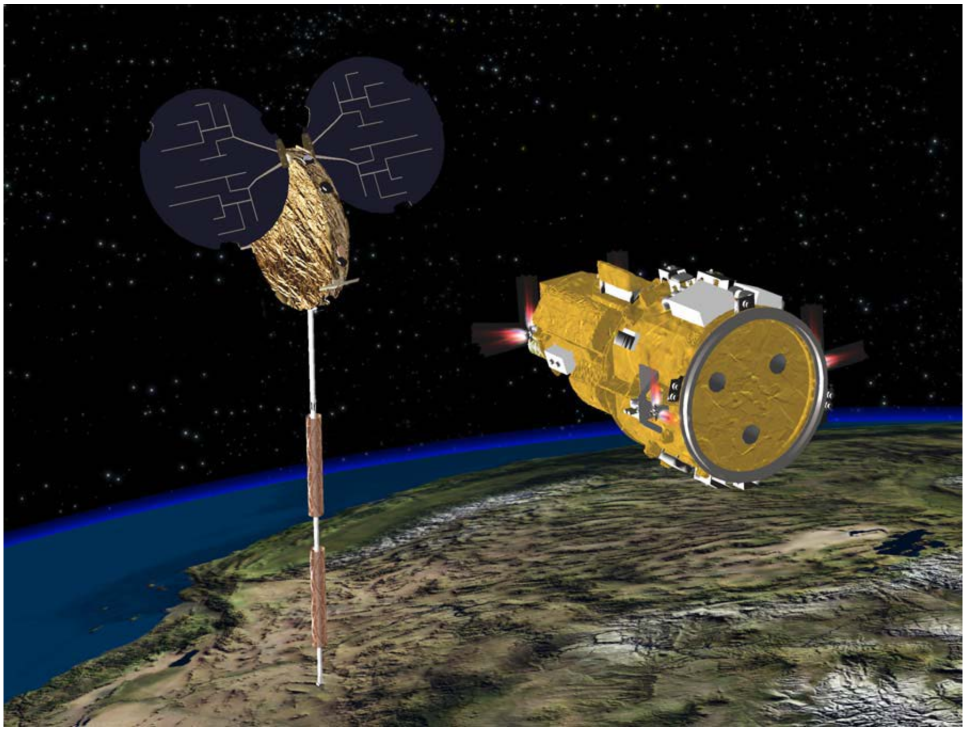


Figure 1 – DART and MUBLCOMM performing ProxOps in space [1]

In the development of aerospace systems, verification and validation (V&V) are often focused on demonstrating that software algorithms and systems will work under nominal conditions. The robustness of the system to off-nominal scenarios is often not tested. Even when system robustness is evaluated, it is difficult to evaluate all possible failure modes. As more missions undertake autonomous operations, there is an increased need for real-time prevention of failures through fault protection. These capabilities are especially necessary for time-critical operations such as rendezvous and ProxOps. Deep space proximity operations applications require advanced autonomy and fault protection due to the significant round-trip light time from Earth.

1.2 Background

Some standardized fault protection nomenclature has been established in the aerospace industry. Also, several key paradigms of fault protection have been developed. This section provides some insight into trends in fault protection practice.

1.2.1 Key Definitions

The NASA Fault Management Handbook defines a failure as “the unacceptable performance of an intended function,” while a fault is defined as “a physical or logical cause, which explains a failure” [2]. This distinction is important because while failures should be avoided, faults are often unavoidable. For example, system designers may not be able to prevent a sensor malfunction (a fault), but they can prevent this malfunction from becoming a catastrophic failure through the use of fault protection (FP). Thus, fault protection systems aim to perform a three-step process called Fault Detection, Isolation, and Recovery (FDIR) in order to prevent failures. Fault detection involves determining that something unexpected has occurred. Fault isolation (also connected to diagnosis) determines the possible source of a fault. Fault recovery is an action taken to attempt to retain or regain control of the system state and mitigate the impact of the fault. These activities often happen simultaneously and can take on various degrees of specificity. FDIR systems avoid the presence of both false positives (non-faults that trigger the detection system) and false negatives (faults that go undetected). These FDIR errors are illustrated in Table 1.

Table 1 – Terminology of true and false positive & negative fault detection [3]

		Fault Detection	
		Positive	Negative
Ground truth	Positive	True positive	False negative
	Negative	False positive	True negative

1.2.2 Rule-Based Fault Protection Paradigm

Over the past decade, the development of FDIR for space missions has advanced significantly. A typical aerospace FDIR system is “a smart embedded system that is able to react to some know[n] events and to select a decision among a predefined set” [4]. Currently, the state-of-the-art in spacecraft FDIR involves using a set of rules that are checked against telemetry. These telemetry monitors are searching for fault symptoms, “ranging from explicitly detected constraint violations, to unexpected hardware behavior, to excessive performance errors, to broken deadlines and more” [5]. Pre-programmed responses are executed when one of these rules is violated [6,7]. For example, if a parameter persistently exceeds its expected range an action is taken by the system, such as transitioning the vehicle into safe mode, stopping all normal mission tasks and turning off all non-essential hardware until ground operators can resolve the fault.

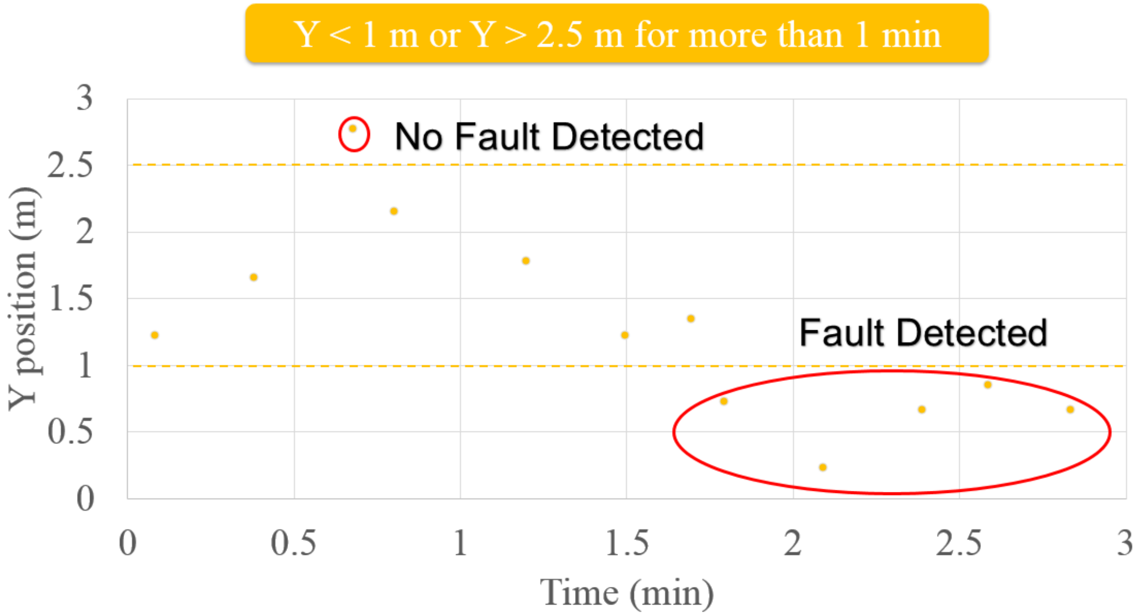


Figure 2 – Example of the rule-based fault protection paradigm

Consider the scenario illustrated in Figure 2, where a sample telemetry dataset representing the Y position of the vehicle in meters is plotted against time in minutes. The rule shown in the yellow box indicates that a fault is detected if the value of Y drops below 1 meter or exceeds 2.5 meters for more than 1 min. Around 0.75 min, a single data point exceeds the limit, however a fault is not detected because the 1 min persistence threshold is not exceeded. Around 1.75 min, the Y value violates the lower limit consistently and triggers a fault detection around 2.75 min.

Of course, the case presented here is a simple example. Many combinations of logical rules are possible, and they are implemented in FDIR systems with increasing complexity. For example, the Dawn mission to explore the asteroid belt makes use of Relative Time Sequences (RTS) that are triggered in response to Telemetry Monitors (TMons) being tripped [8]. Each RTS can then enable, disable, or start other RTS's and can enable or disable TMons. Dawn contains over 250 RTS and over 250 TMons, all of which are documented in an Excel spreadsheet. The system is so complex that a model-based SPIN model checker was used for V&V to ensure that it was constructed with logical soundness [8].

Another key component of the rule-based FP paradigm is the use of "safe mode". If something goes wrong that cannot be solved onboard, the spacecraft will autonomously transition to safe mode. Safe mode involves stopping all normal mission tasks, turning off all non-essential hardware to conserve power, orienting the spacecraft in a favorable attitude (for thermal concerns, power generation, and communications), and waiting for commands from the ground to return the spacecraft to normal operation mode. These functions are shown in Figure 3.

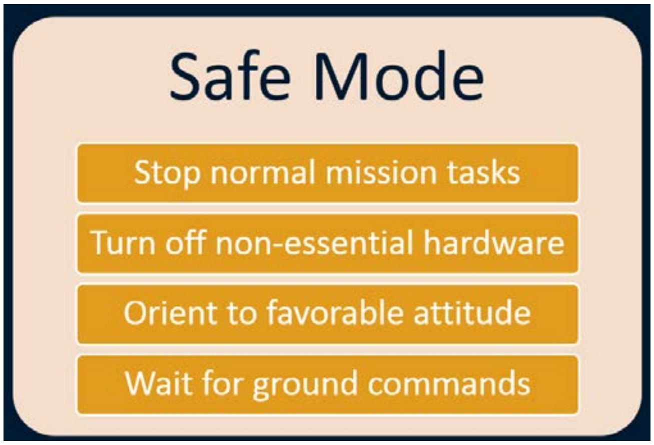

Safe Mode
Stop normal mission tasks
Turn off non-essential hardware
Orient to favorable attitude
Wait for ground commands

Figure 3 – Typical changes to spacecraft functionality during Safe Mode

A spacecraft should be able to survive in safe mode, but it will not be able to continue its mission until the problem is resolved on the ground and normal operations are restored. In this paradigm, detection of fault symptoms is usually mapped directly to a pre-determined response sequence. In other words, fault diagnosis is performed by FP engineers at design time rather than by the system on-board in real-time. A typical sequence for fault protection response is shown in Figure 4, with an example given for a camera power fault.

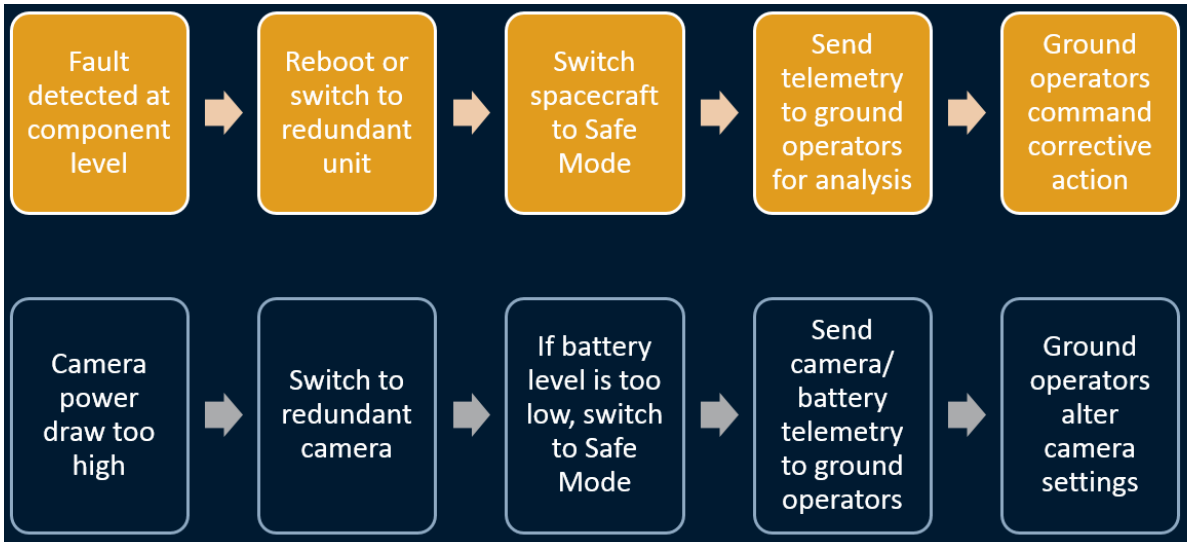

Fault detected at component level
Reboot or switch to redundant unit
Switch spacecraft to Safe Mode
Send telemetry to ground operators for analysis
Ground operators command corrective action
Camera power draw too high
Switch to redundant camera
If battery level is too low, switch to Safe Mode
Send camera/ battery telemetry to ground operators
Ground operators alter camera settings

Figure 4 – Typical Fault Protection response sequence

Traditional Fault Protection proceeds in a hierarchical fashion [9], as shown in Table 2. First, a fault is detected inside a particular spacecraft hardware or software component and local corrections are attempted (Level 0). Next, the fault propagates up to subsystem software (Level 1). If subsystem software cannot resolve the fault, or if several faults occur simultaneously, then the system will be reconfigured (Level 2), such as switching to a full suite of redundant hardware. Failure of system control hardware or software, such as the flight computer that houses fault protection (Level 3) will also result in system reconfiguration. If multiple Level 2 or 3 failures or any major overall system failure occurs (Level 4), the spacecraft will transition into safe mode and rely on the ground team to investigate the fault and provide a resolution.

Table 2 – Traditional FDIR Response Structure, adapted from [9]

Fault Level	Response Level	Description	Impact	Fault Detection	System Recovery
Level 0	Unit Internal	Failure without effects on performance	No impact on system performance	Local in-unit checks	Local in unit command retry, unit reboot
Level 1	Subsystem Software	Unit failure or subsystem performance degradation	Degraded performance of subsystem	In respective subsystem; limit checking of unit parameters	Switch to redundant unit, command retry, reboot
Level 2	System Reconfiguration	Subsystem failure not recovered by previous levels	Performance loss of subsystem	Several alarms from unit level 0 consistency checks	Switch to redundant side, command retry
Level 3	System Control Software	Failure of equipment involved in FDIR	Performance loss of subsystem	Faults on FDIR units	Switch to redundant side, command retry
Level 4	Flight Operations System on Ground	Multiple level 2/3 failures, major overall system failure	System performance loss; mission interruption	Several alarms from level 2 & 3, Hardware alarms	Change to safe mode, wait for ground to recover system

In addition, it should be noted that for critical events such as orbit insertion, flybys, entry, descent, and landing, or rendezvous & ProxOps, safe mode is often “disabled” because the spacecraft must maintain a “fail-operational” rather than “fail-safe” response from onboard FDIR systems [2]. The critical event sequences *must* be completed and cannot simply be paused if a fault occurs because of the risk of loss of vehicle or mission. Typically, autonomous FDIR systems will put the vehicle into safe mode when a fault is detected and await input from ground operators, as occurred with the Japanese Venus Climate Orbiter *Akatsuki* during orbit insertion [10]. If the vehicle enters safe mode during mission-critical times, such as *Akatsuki*’s orbit insertion, certain objectives may not be met. In the case of *Aktsuki*, the spacecraft failed to enter orbit. It is the presence of such critical events that motivates a need for greater FP autonomy, especially in highly complex interplanetary missions. The rule-based paradigm often cannot handle critical events in a fail-operational manner when long signal time delays make ground controllers unable to respond to faults in a timely manner [9].

The rule-based fault protection paradigm has been used successfully for decades and is fairly straightforward to understand and test. Rule-based systems can be pre-programmed without anticipating every possible scenario. The rule-based paradigm is the current standard for spacecraft fault protection and is used extensively in industry because of its utility and ease of implementation. However, several problems with this paradigm have been identified by FDIR practitioners. Adding logical expressiveness to enable more advanced rules (such as adding different kinds of logical operators or enabling multiple logical conditions in each rule) adds complexity to the system, reduces ease of understanding for developers and reviewers, and vastly increases testing time [6,7]. An example of a complex rule from [6] is shown here:

```
OBS_NODE != MODE_ASCENT && TWTA_VOLT_EU > 5.0 && RF_EPC_ST
> 0x4A && RF_TWT_ST > 0x4A && (( BATT_PRES1_EU > 650.0 &&
BATT_PRES1_ST == MUX_AD_OK ) || ( BATT_PRESS2_EU > 650.0 &&
BATT_PRESS2_ST == MUX_AD_OK ))
```

Also, *ad hoc* approaches can result in gaps and inefficiencies in overall FDIR design [11]. In real mission scenarios, it can be difficult to determine the meaning or the cause when a fault detection rule is triggered, and it is not always clear how to respond to rule triggers. For example, as one FP practitioner has said in [5]:

> *"Fault management systems generally respond to problematic "errors", as undesirable deviations, but there is little discrimination between deviations from*

> *modeled behavior, deviations from predictions, deviations from objectives, or deviations from "nominal" or "safe" conditions (neither of which is well defined). Similarly, it is not clear, when a threshold is tripped, whether this reflects an assessment of system state (e.g., a device has failed), an objective violation (e.g., the device cannot perform some required function), or a control decision (e.g., something must be done about the failure). In conventional designs, it could be any or all of these, conflated and demoted to an inscrutable act of arithmetic."*

Another FP researcher [9] states that:

> *"Literature reports conventional FDIR methods suffering from significant shortcomings, like often missing isolation of faults and failures on-board, only partial observability of the actual system status and no on-board knowledge at all about the general operational capabilities of the system."*

Finally, although FDIR responsibilities are similar on various missions, the implementation of FDIR systems (including logical rule statements) often does not carry over from previous missions [6]. While principles of FDIR design and lessons learned do sometimes carry over, spacecraft design teams tend to develop custom sets of FDIR rules and rule-based systems from scratch based upon the specific needs and requirements of each new mission.

1.2.3 Model-Based Fault Protection Paradigm

The use of a new paradigm called model-based fault protection has been explored and implemented in some scenarios. This paradigm uses a model of the flight system's behavior and selects the “state” of the system based on telemetry. Model-based architectures are necessarily state-based, because all models used for system control must be state-based [5]. Using behavior models of system state allows more complex and insightful decisions to be made by the autonomous system about fault determination and response.

One space systems engineering team at the Jet Propulsion Laboratory (JPL) has begun to analyze the FDIR problem in depth using model-based systems engineering approaches [11]. This team has developed an FDIR architecture using the SysML modeling language. Although this architecture is used for identifying, evaluating, and managing failure modes during the design and V&V phases, the implementation of FDIR for flight software (FSW) does not stem directly from the architecture. The example in Figure 5 shows a model-based architecture of Guidance, Navigation, & Control (GN&C) failure modes for a generic Earth-orbiting satellite mission. Here GN&C activities (magenta) are connected to various failure modes (red), which could be related to a number of different possible causes (blue), and may result in various subsystem effects (yellow) and system-level effects (orange). More recently, JPL has developed an ontology (formal technical vocabulary) for this kind of analysis, one important step toward enabling automated creation of traditional FP analyses such as Failure Modes, Effects, and Criticality Analysis (FMECA), Fault Tree Analysis (FTA), and Fault Containment Region (FCR) diagrams [12]. As mentioned in the previous section, the Dawn mission team at JPL also made use of a model-based logic checker during V&V of its FP system. A model of system behavior is required in order to perform checks of logic for consistency against specifications [8].

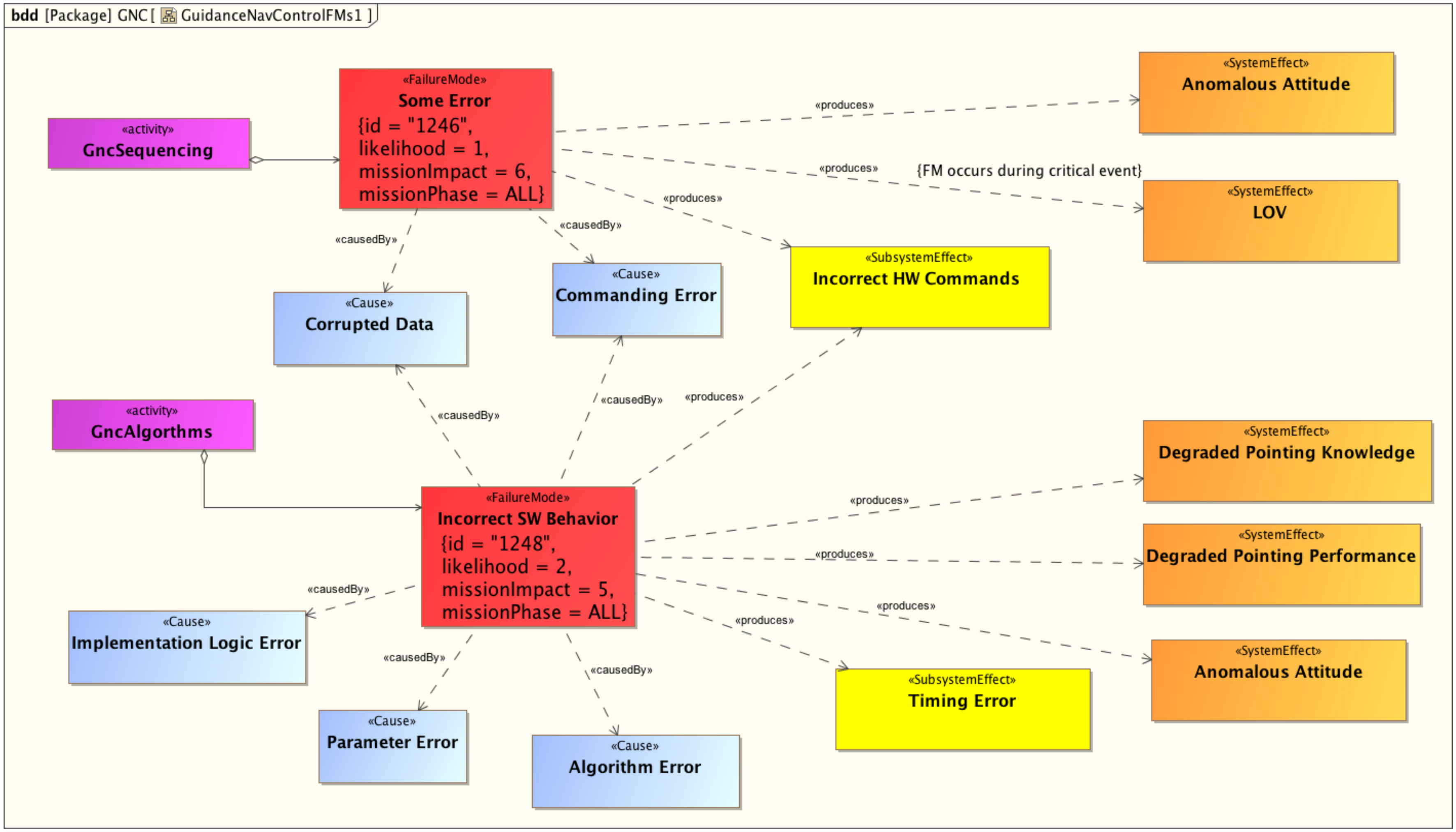


Figure 5 – Example model-based fault protection architecture [11]

1.2.4 State Machine Logic and Applications

The “state” of a system includes any “aspects of the system that we care about for the purposes of control” [13]. Traditionally, state variables have included continuous physical parameters such as position, velocity, attitude, temperature, and pressure. However, state variables can also include discrete quantities such as operating modes and device health. These discrete states can then be represented as state machines. State machines, or state charts, are a specific model-based tool used to represent complex logical relationships. They provide a visual block-diagram development that is fairly straightforward to understand and can be applied to fault protection [13]. Each block represents a specific state or sub-state of the system, and arrows between blocks represent transitions between states. A logical condition is associated with each transition, and if the condition associated with the transition becomes true, then the active state of the diagram will move from one state to another.

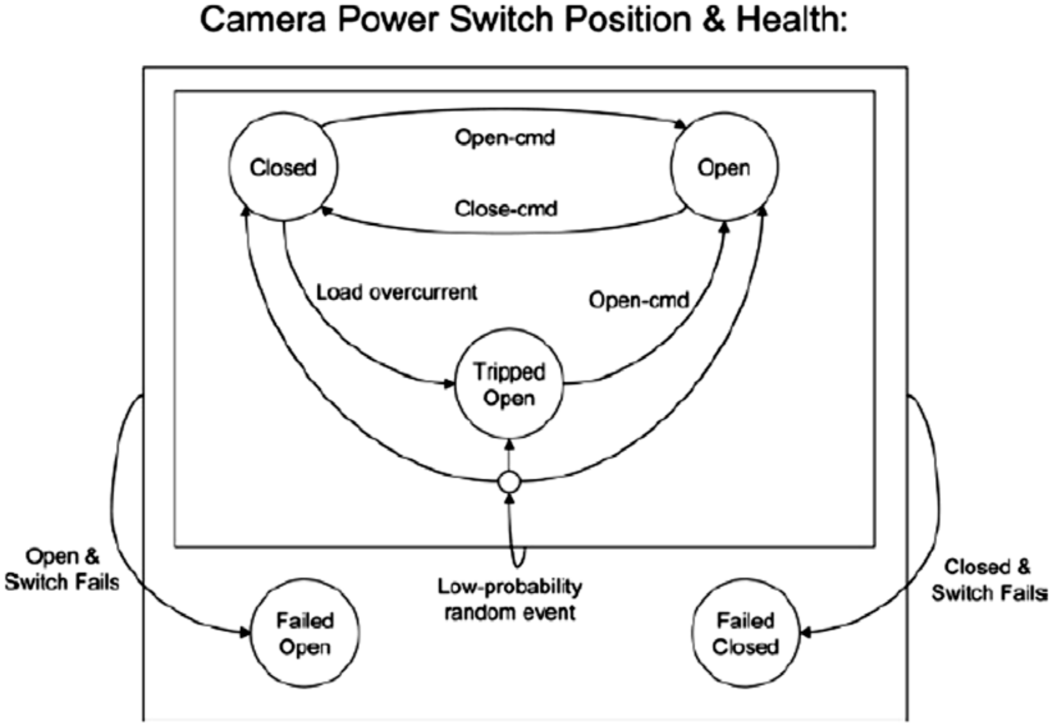


Figure 6 – Example usage of state machines for fault protection [13]

The example state machine in Figure 6 shows the possible states of a camera power switch. There are two primary states of the switch, “Open” and “Closed”. Transitions between the states are activated when an “Open-cmd” or “Close-cmd” command is sent to the camera. Additional states and transitions such as “Tripped Open” (a fault state) and “Load overcurrent” (a root cause of the fault state) are added to illustrate a known fault condition of the system. Finally, fault states “Failed Open” and “Failed Closed” are added into the system, showing how the state machine can be used to implement fault detection.

State machine representations may be significantly simpler than the actual physical or software processes they represent, which is why they are considered models. However, a state machine for FDIR purposes can be developed in a way that represents all possible states relevant to mission success. FP systems expressed in terms of system state will be better able to protect the system in question [5]. Knowledge of the state is not the same as the state itself, and the status of a state machine representation at any time is only as accurate as the information that is provided to it. If input data is outdated or incorrect, the active state chosen by the state machine representation may be outdated or incorrect as well. Proper state knowledge always includes this resulting uncertainty [5].

Within MATLAB/Simulink, the Stateflow toolbox [14] provides a simple graphical interface for developing state machines, which can be used to represent the current state of different vehicle hardware or software components. Stateflow charts can be very simple, representing only a few possibilities, or they can involve numerous complicated nested sets of states. Transitions are indicated by blue arrows with Boolean conditions; if a condition registers as true, the transition will be activated to move from one state (or substate) to another. "Default" transitions specify the initial conditions of the diagram and are indicated by an arrow beginning at a dot and ending at the initial state or substate. Stateflow animates active states and transitions in dark blue during simulation so that the developer can monitor the simulation as it runs for debugging and confirmation that the chart is properly constructed.

An example Stateflow chart representing a thruster controller developed for the Prox-1 small satellite mission [15] is shown in Figure 7. This chart contains three states (Startup, ThrustOff, ThrustOn), transition conditions between the states, and an embedded function "fuel_check" written in MATLAB syntax. The chart determines whether the thruster should be on or off based on whether the controller has received a command to fire ("ready"), the amount of time commanded ("time"), and whether sufficient fuel is available (determined by the output "enough_fuel" from the MATLAB function).

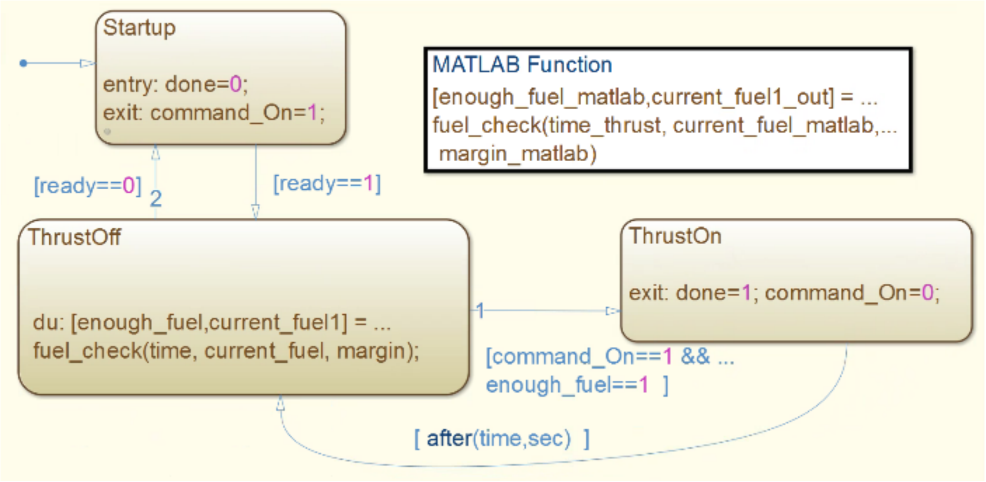

Startup
entry: done=0;
exit: command_On=1;
MATLAB Function
[enough_fuel_matlab,current_fuel1_out] = ...
fuel_check(time_thrust, current_fuel_matlab,...
margin_matlab)
[ready==0]
2
[ready==1]
ThrustOff
du: [enough_fuel,current_fuel1] = ...
fuel_check(time, current_fuel, margin);
ThrustOn
exit: done=1; command_On=0;
1
[command_On==1 && ...
enough_fuel==1]
[after(time,sec)]

Figure 7 – Sample Stateflow chart representing a thruster controller [15]

The Stateflow chart is integrated within a Simulink model as a Stateflow block with inputs and outputs, as shown in Figure 8. Stateflow logic allows complex decisions to be made in a hierarchical way, where conditions and logical states in individual spacecraft components, FDIR algorithms, and higher level "master" FSW mode logic all influence one another.

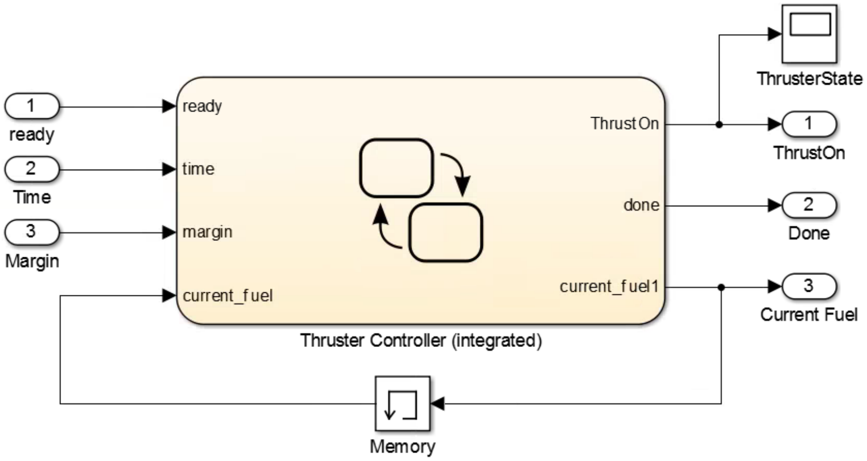
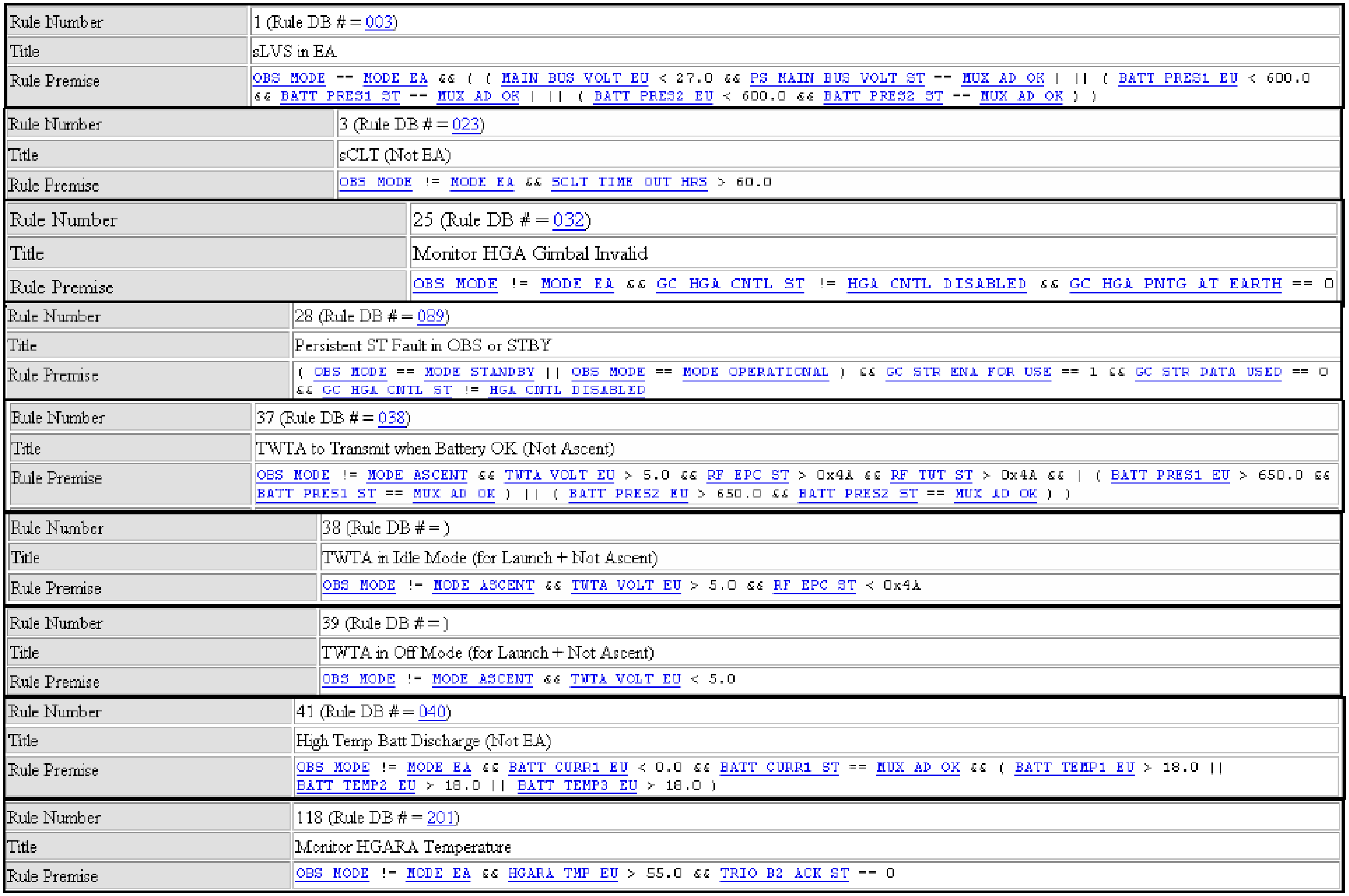


Figure 8 – Example of Stateflow block integration in Simulink [15]

Several space mission teams have made use of the Stateflow toolbox in MATLAB/Simulink [14] to develop FDIR algorithms as state machines. These algorithms are then converted into C/C++ code using a process called “autocoding.” Missions that have autocoded FDIR algorithms from MATLAB/Simulink into FSW include Deep Space 1 [16] and Deep Impact [17]. NASA’s Johnson Space Center has used MATLAB/Simulink, including Stateflow, to develop algorithms for GN&C, which are later autocoded into FSW [18]. JPL has also developed an open-source Statechart Autocoder that converts state machines from the Unified Modeling Language (UML) format to C/C++ [19]. Stateflow has been used to evaluate errors in FDIR algorithms during spacecraft system V&V [20]. Another FDIR architecture developed with Stateflow uses model-based design techniques to bring V&V earlier in the design cycle by providing a link between subsystem design and FDIR design [21]. Finally, JPL has developed an ontology for enabling formal description and specification of state-based system behavior, which includes modeling using state machines [22].

State machines offer several advantages over the rule-based FP paradigm. One significant advantage is the generation of a graphical product that is easier for designers, peer reviewers, and managers to understand and review. Other advantages include ease of accounting for subsystem interdependencies and implementing sequences with several decision points and/or path-dependent responses. The Johns Hopkins Applied Physics Lab (APL) conducted a formal trade study to determine whether their “ExecSpec” state-based fault protection system [6,7] or a more traditional rule-based system was more advantageous using the Solar Probe Plus mission as a case study [23]. They found that both methods were able to equivalently express all desired fault protection rules but that the state machine system is favored based on some of the advantages mentioned above. However, APL ultimately chose to continue using the rule-based system due to its extensive flight heritage. A direct comparison showing the same FP logic implemented in both a rule-based and a state-based system is shown in Figure 9 and Figure 10, as originally published by APL [6]. This logic was used to monitor a radio frequency amplifier in the Solar TErrestrial RElations Observatory (STERO). Note how much more readily the graphical state machine can be interpreted and checked for accuracy.



Rule Number	1 (Rule DB # = 003)
Title	sLVS in EA
Rule Premise	OBS_MODE -- MODE_EA && ((MAIN_BUS_VOLT_EU < 27.0 && PS_MAIN_BUS_VOLT_ST -- MUX_AD_OK | || (BATT_PRES1_EU < 600.0 && BATT_PRES1_ST -- MUX_AD_OK | || (BATT_PRES2_EU < 600.0 && BATT_PRES2_ST -- MUX_AD_OK))
Rule Number	3 (Rule DB # = 023)
Title	sCLT (Not EA)
Rule Premise	OBS_MODE != MODE_EA && SCLT_TIME_OUT_HRS > 60.0
Rule Number	25 (Rule DB # = 032)
Title	Monitor HGA Gimbal Invalid
Rule Premise	OBS_MODE != MODE_EA && GC_HGA_CNTL_ST != HGA_CNTL_DISABLED && GC_HGA_PNTG_AT_EARTH == 0
Rule Number	28 (Rule DB # = 089)
Title	Persistent ST Fault in OBS or STBY
Rule Premise	(OBS_MODE == MODE_STANDBY || OBS_MODE == MODE_OPERATIONAL) && GC_STR_ENA_FOR_USE == 1 && GC_STR_DATA_USED == 0 && GC_HGA_CNTL_ST != HGA_CNTL_DISABLED
Rule Number	37 (Rule DB # = 038)
Title	TWTA to Transmit when Battery OK (Not Ascent)
Rule Premise	OBS_MODE != MODE_ASCENT && TWTA_VOLT_EU > 5.0 && RF_EPC_ST > 0x4A && RF_TUT_ST > 0x4A && | (BATT_PRES1_EU > 650.0 && BATT_PRES1_ST == MUX_AD_OK) || (BATT_PRES2_EU > 650.0 && BATT_PRES2_ST == MUX_AD_OK))
Rule Number	38 (Rule DB # =)
Title	TWTA in Idle Mode (for Launch + Not Ascent)
Rule Premise	OBS_MODE !- MODE_ASCENT && TWTA_VOLT_EU > 5.0 && RF_EPC_ST < 0x4A
Rule Number	39 (Rule DB # =)
Title	TWTA in Off Mode (for Launch + Not Ascent)
Rule Premise	OBS_MODE !- MODE_ASCENT && TWTA_VOLT_EU < 5.0
Rule Number	41 (Rule DB # = 040)
Title	High Temp Batt Discharge (Not EA)
Rule Premise	OBS_MODE != MODE_EA && BATT_CURR1_EU < 0.0 && BATT_CURR1_ST == MUX_AD_OK && (BATT_TEMP1_EU > 18.0 || BATT_TEMP2_EU > 18.0 || BATT_TEMP3_EU > 18.0)
Rule Number	118 (Rule DB # = 201)
Title	Monitor HGARA Temperature
Rule Premise	OBS_MODE !- MODE_EA && HGARA_TMP_EU > 55.0 && TRIO_B2_ACK_ST -- 0

Figure 9 – Rule-Based Fault Management of STEREO Radio Frequency Amplifier [6]

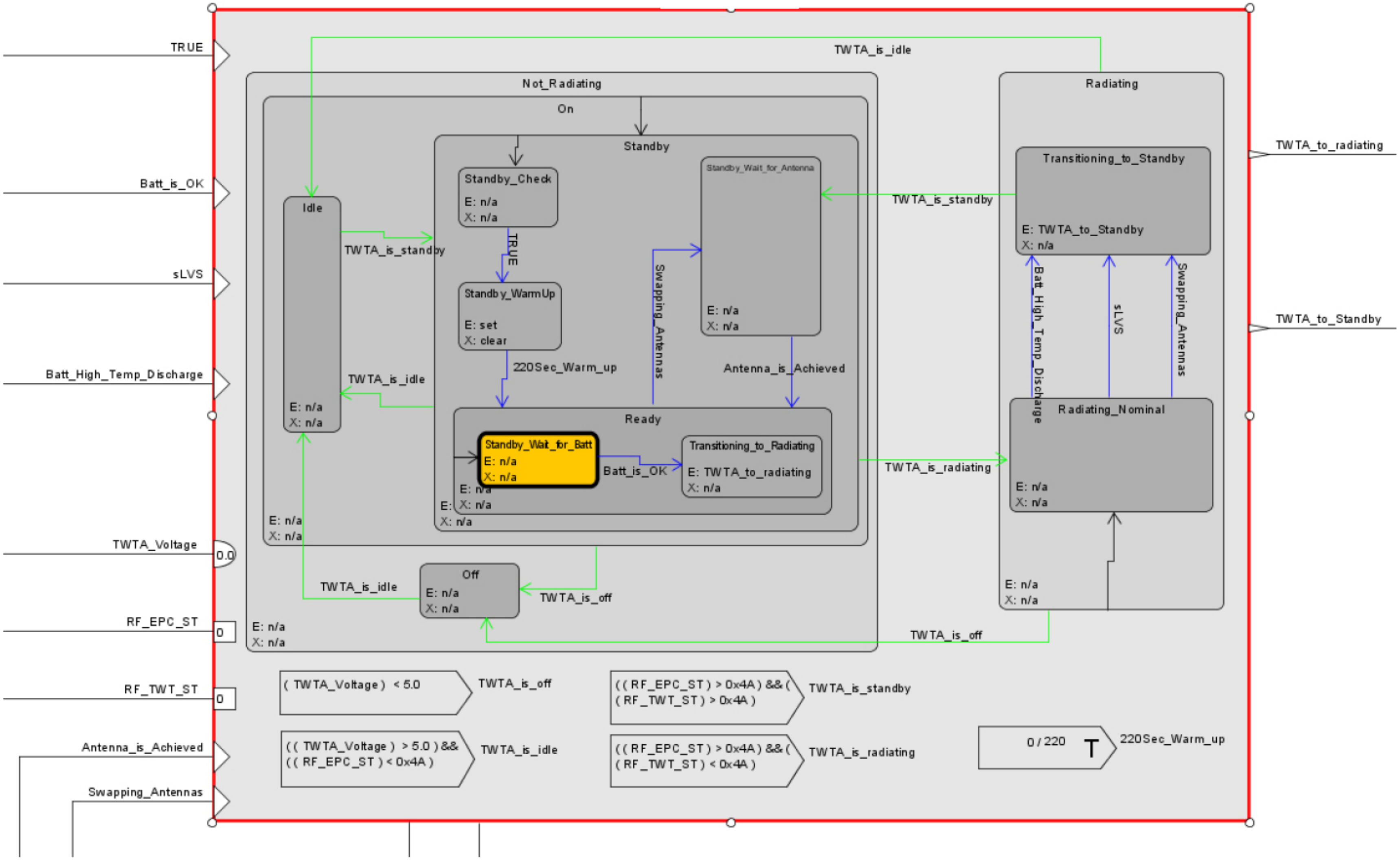


Figure 10 – State-Based Fault Management of STEREO Radio Frequency Amplifier [6]

1.2.5 Goal-Based Autonomy Paradigm

Although this dissertation will not utilize it directly, an introduction to goal-based autonomy is appropriate at this point. FP is actually a subset of system autonomy, and goal-based methods are an area of current research in spacecraft autonomy. A goal-based architecture uses objectives to control an autonomous system rather than directly commanding all actions in sequences of linear commands. An objective "is nothing more or less than a model of desired changes of state in the system under control" so that "a goal-based architecture is necessarily state-based as well" [5]. One key goal-based autonomy platform is Remote Agent [24], which was deployed as a technology demonstration (not as the primary control software) on the Deep Space 1 mission. Although goal-based architectures can be used for FDIR, they are inherently full-system autonomy architectures and are thus out of the scope of this dissertation.

1.2.6 On-Board Model-Based Fault Diagnosis

Fault diagnosis is usually performed by FDIR engineers at design time, and the detection of a specific symptom is directly mapped to the appropriate response for the pre-diagnosed fault. Though not typical for space missions, on-board fault diagnosis has been an area of research since the 1990s. One researcher [5] states that:

> *"Error monitors in general tend to have…problems when errors are not interpreted through models or correlated and reconciled with other evidence. When such a diagnostic layer is absent, and responses are triggered directly by monitor events, it becomes hard to put one's finger on what exactly a system believes it is responding to."*

This reflects the *ad hoc* nature of pre-programmed responses triggered directly by fault symptom detection. According to one FDIR overview paper, model-based fault diagnosis is considered a structured and mature field of research and many methods have been proposed and discussed in the control community [9]. One extensive survey of model-based fault diagnosis methods discusses various mathematical control/estimation methods for aeronautical vehicles [25]. Remote Agent featured model-based "mode identification" and "mode reconfiguration" for fault diagnosis, which identify components whose failures explain detected anomalies [24]. Cassini's Attitude Control Fault Protection is one of the few examples of a system where on-board fault diagnosis was performed in-flight [26,27]. One method for on-board diagnosis that has been used in research studies is called constraint suspension. It has been used to diagnose which component of a system is faulty [28,29].

1.3 Contributions of This Investigation

This dissertation analyzes the FDIR problem for aerospace vehicles in the generic sense, focusing on the GN&C subsystem. The complexity, ubiquity, and autonomous nature of GN&C makes it a relevant example case for exploring fault protection advances that apply across all subsystems for many different aerospace applications. This investigation results in three key contributions that advance the state-of-the art in aerospace FDIR.

1.3.1 Generic Fault Protection Architecture

Contribution 1: Develop a generic, modular, and portable FDIR architecture that can be used for the GN&C subsystem on a wide variety of aerospace missions and vehicles.

Many missions build custom fault protection systems from the ground up while adapting principles from previous missions. Also, FDIR is often a specialized task performed by systems engineers after mission and vehicle design has been completed. This contribution aims to develop a generic architecture that is applicable to any type of aerospace vehicle or mission. A modular architecture is built with components that can be easily added, removed, or rearranged, allowing system design and FDIR design to be completed in parallel. This moves FDIR development earlier in the flight project lifecycle, thus allowing FDIR to influence system design and to be validated during system-level testing. The FDIR architecture is also portable to FSW. In other words, it is straightforward to convert the design implementation derived from the architecture for a particular mission to code that is used onboard the vehicle. The scope of the architecture in this study focuses on GN&C, but it is be presented in a manner that is applicable to all vehicle subsystems.

1.3.2 On-Board Model-Based Diagnosis

Contribution 2: Use a state machine formulation of the generic FDIR architecture to perform on-board model-based fault diagnosis.

Although the model-based paradigm for fault protection has been explored by industry, it has not yet been widely adopted. This study focuses on the state machine approach to model-based FDIR, which has been used in several flight projects and research studies because it is intuitive, logic-based, and simple to interpret visually. State-of-the-art fault protection practice involves monitoring telemetry or sensor data of an aerospace system for symptoms of faults such as upper or lower limit violations. Diagnosis is usually performed by FDIR engineers at design time, and the detection of a specific symptom is directly mapped to the appropriate response for the pre-diagnosed fault. This study advances the state-of-the-art in state-based fault protection by developing an on-board diagnostic system that assesses symptoms, isolates fault sources (while accounting for uncertainty), and selects corrective actions based on models of system behavior.

1.3.3 UAV and Space-Based ProxOps Applications

Contribution 3: Adapt the state machine FDIR architecture for terrestrial unmanned aerial vehicle and space-based proximity operations fault protection applications.

To demonstrate the applicability of the state machine FDIR architecture to realistic scenarios from a wide variety of aerospace vehicles and missions, two applications are explored in very different domains. Each of these applications makes use of GN&C and undergoes V&V. The first is a terrestrial application involving the use of multi-rotor unmanned aerial vehicles (UAVs) for 3D scanning and mapping. This application undergoes V&V through flight testing. The second application is a space-based scenario involving automated rendezvous and ProxOps for orbital capture in a Mars Sample Return mission. This application undergoes V&V through software-in-the-loop (SITL) testing in a flight-like spacecraft software simulation environment.

CHAPTER 2. THEORY: GENERIC FAULT PROTECTION ARCHITECTURE

The study presented here advances the state-of-the-art in FDIR and builds on previous work by bringing together capabilities such as model-based design and autocoding to FSW into a single generic, modular FDIR architecture that is portable to FSW. This architecture also features model-based on-board fault diagnosis using state machines. While most previous FDIR implementations have involved large, high-resource missions with custom-built FDIR, the proposed architecture is designed to be applicable to a wide variety of platforms and missions. The architecture also allows alternate configurations that enable testing of various scenarios.

2.1 FDIR Architecture Concept Overview

The FDIR architecture collects data from the vehicle which is used to determine the likely state of the vehicle. This state can be classified as either "fault" or "no fault" based on how the decision logic is structured. The architecture isolates faults by performing diagnosis to determine their precise source and performs preventative actions to recover from faults before they become mission-critical failures. Outputs from the architecture can either send commands to the vehicle autonomously or notify ground operators to take corrective action.

2.1.1 FDIR Architecture Requirements

At the beginning of this study, high-level requirements were identified to guide the development of this FDIR architecture: it must be generic, modular, and portable to FSW. A generic architecture is applicable to any type of aerospace vehicle or mission. A modular architecture allows components to be easily rearranged, added, or removed. A portable architecture means that it is straightforward to convert the design implementation derived from the architecture for a particular mission to code that is used onboard the vehicle.

In addition to the three primary requirements of generic, modular, and portable, additional guidelines for the architecture were also established. V&V of the FDIR architecture should assess its capability to meet the following goals. It should detect and possibly correct software and hardware faults at multiple levels: component, subsystem, and system. These faults may include sensor/actuator malfunctions, errors or degradation, improper controller gain settings, non-convergence of GN&C algorithms, and avionics software or processor hardware faults. The architecture should detect and avoid mission-level failure modes, such as vehicle collision or uncontrolled behavior that renders the mission objectives unattainable. Faults should be detected, diagnosed, and corrected in real-time onboard as they occur, not in post-processing or by operators examining data on the ground.

The architecture should utilize model-based decision logic rather than the rule-based paradigm. Model-based approaches allow the system to select the best course of action when multiple options exist. Finally, the architecture should demonstrate FP logic that allows the system to avoid aborts by responding to correctible errors in real-time to meet mission objectives. Because different missions have different risk classes (A,B,C,D) [30], the architecture should scale to meet the requirements of the mission.

2.1.2 Development Environment in MATLAB/Simulink

The architecture discussed in this study leverages the development of a Six-Degree-of-Freedom (6DOF) simulation environment for the Prox-1 small satellite mission and several other projects at Georgia Tech. The simulation environment models in-flight conditions of actual vehicles and missions and contains environment and hardware models with configurable settings. The original purpose of this MATLAB/Simulink platform was for Prox-1 GN&C algorithm integration and testing [15,31] and it has been used for feasibility studies of constellations of CubeSats at Mars [32] and a Processor-in-the-Loop testbed for high-fidelity testing of avionics boards for relative proximity operations called SoftSim6D [33]. Functionality has been added to the simulation environment that can be applied generally to aerospace mission scenarios to test a variety of FDIR algorithms and mission architectures.

2.1.3 Enabling Advances in FDIR

The primary area of applicability of this architecture to the NASA Technology Area Breakdown Structure is element 4.5.1 System Health Management under section 4.5 System Level Autonomy within Technology Area 04: Robotics and Autonomous Systems. System health management "monitors, predicts, detects, and diagnoses faults and accommodates or mitigates the effects either on-board or through telemetry processing on the ground" [34]. The FDIR architecture results in on-board real-time system health management software and will address many of the desired technical capabilities of element 4.5.1. For example, complex logic allows the FDIR architecture to include prognostic and diagnostic components as an integral part of the system. The logic is also able to take complicated vehicle states into account to avoid false positives when faults are not present and false negatives when faults are present. It may even be used to anticipate faults and adapt to new situations that do not have pre-programmed responses. Finally, this study advances paradigm-shifting model-based approaches for FDIR that can be easily transitioned to FSW and validated using SITL and flight testing for V&V.

2.2 FDIR Architecture Characteristics

This section focuses on explaining how the architecture meets the primary requirements of generic, modular, and portable. The architecture itself are described in the following section.

2.2.1 Generic

A generic architecture is applicable to any type of aerospace vehicle or mission. The FDIR architecture is comprised primarily of five generic diagrams that are described in the following section. The MATLAB/Simulink simulation environment in which the architecture is developed allows setting vehicle parameters including physical dimensions and trajectory. It is applicable to a multitude of possible mission scenarios and permits alternate configurations, such as individual vehicles or multiple cooperative or non-cooperative vehicles. The simulation environment also contains generic modules for commonly used components such as sensors and actuators. The simulation environment has previously been adapted for use with many scenarios, missions, and vehicles, including the Prox-1 small satellite mission [15,31], various ProxOps scenarios with hardware such as a modular attitude determination system CubeSat avionics board [33], and a Mars communication relay CubeSat constellation [32]. Each of the five diagrams described in the following section are implemented without focusing on any particular application or vehicle. Chapters 3 and 4 demonstrate how the architecture can be adapted for two distinct and very different applications. While the generic architecture presented here is focused particularly on FDIR for faults related to the GN&C subsystem, the same principles and design can be applied to any other faults and subsystems on an aerospace vehicle.

2.2.2 Modular

A modular architecture allows components to be easily added, removed, or rearranged. The visual block diagram environment offered by MATLAB/Simulink can be altered and reconfigured easily and allows for testing of many combinations of software modules and hardware components. For example, the investigator could replace the sensor/actuator suite and GN&C software modules. An example of such a reconfiguration is shown in Figure 11 and Figure 12, where an image generation (ImageGen) block is added to the generic sensor suite. This block allows generation of simulated images from visual and infrared sensors used during ProxOps.

Also, various initial conditions, environmental scenarios, and physical vehicle properties can be easily redefined in a MATLAB initialization script and edited or rearranged in Simulink. These include spacecraft orbit and attitude dynamics, spacecraft properties such as mass properties and outer mold line, relative dynamics for multiple spacecraft, sensor and actuator properties such as field of view and resolution, GN&C software components, and central body or environment properties. Parameters for FDIR algorithms can also be adjusted, such as fault injection times, wait times, and trigger thresholds. The five diagrams described in the following sections can also be easily adjusted and rearranged to adapt them for various vehicles and missions.

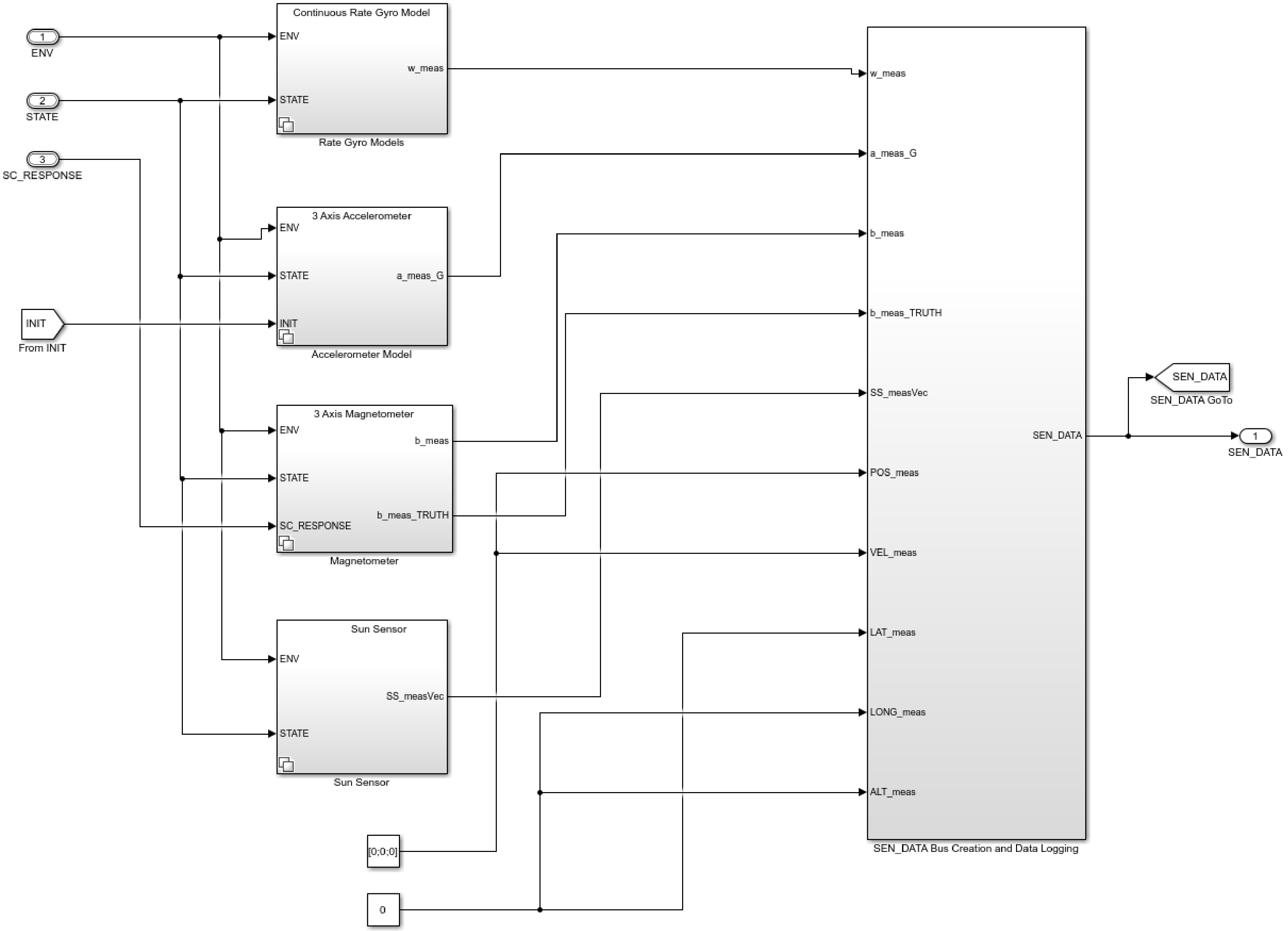


Figure 11 – Generic sensor suite (before reconfiguration)

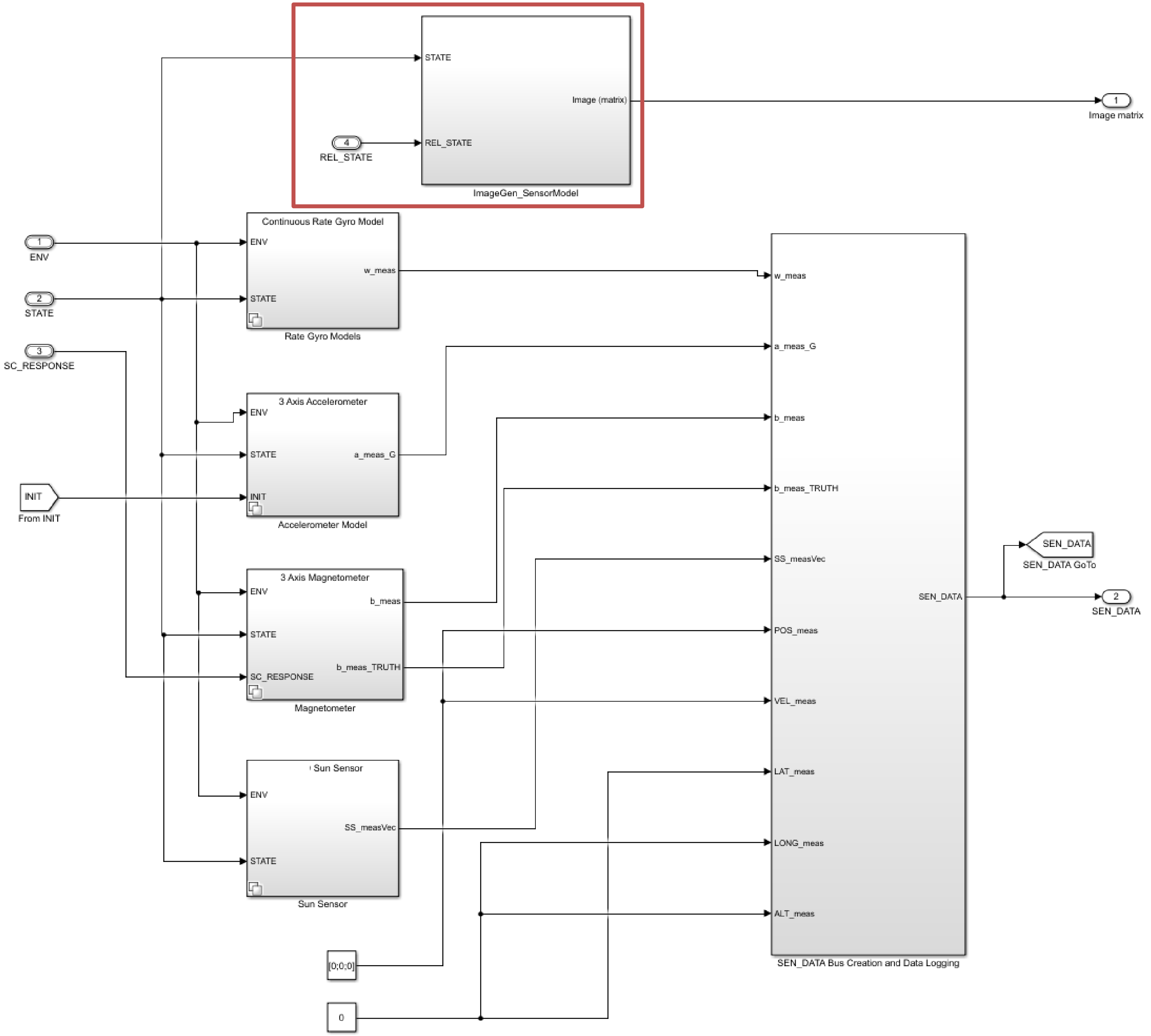


Figure 12 – Generic sensor suite (after reconfiguration) with ImageGen Sensor Model added

2.2.3 Portable

A portable architecture allows straightforward conversion of its design implementation for a particular mission to code that is used onboard the vehicle. The FDIR architecture allows rapid transition from development to flight. The computational requirements of the FDIR architecture match the capability generally available on flight processors. The architecture has the ability to make the kinds of complex decisions normally required for autonomous FSW and is evaluated by testing its response to realistic conditions rather than “canned” scenarios. It is well integrated with other hardware and software components, allowing new components to be quickly evaluated. Finally, the architecture features the capability to easily convert the architecture into FSW code via autocoding, a process which has been used with the Prox-1 mission as described in [15]. In this process, algorithms developed in MATLAB/Simulink are converted to C code and integrated with other FSW code in C. Autocode performance is validated via a “day in the life” test in a testbed using flight hardware. Although the autocoding process is not demonstrated directly in this study, technical memos written by the Prox-1 team are included in Appendix A to provide guidance for future researchers or engineers desiring to reproduce it. Many FDIR algorithms have been developed by other researchers for hardware and software faults, and the architecture developed in this work enables these algorithms to be rigorously tested and implemented. This will greatly facilitate the transition of new FDIR algorithms from concept design to implementation.

2.3 Generic Architecture Diagrams

The FDIR architecture presented in this study is made up primarily of five generic diagrams: a GN&C subsystem taxonomy, fault tree analysis, functional state machine, diagnostic state machine, and FDIR architecture block diagram. Each of these diagrams is described in detail in this section.

2.3.1 Generic GN&C Subsystem Taxonomy

To prepare for the development of the FDIR architecture, it is useful to develop a generic taxonomy of aerospace vehicle subsystems. Subsystems such as telemetry, tracking & control (TT&C), electrical power subsystem (EPS), structures and mechanical systems (S&MS), and GN&C each have many hardware and software components that could manifest faults. Because aerospace vehicle systems are quite complicated, it is useful for FDIR designers to focus more effort on subsystems that are historically prone to fault by examining anomaly trends [35]. One study by the Aerospace Corporation found that among twenty Earth-orbiting vehicles, GN&C accounted for 40% of failures, EPS accounted for 40%, TT&C accounted for 10%, and S&MS accounted for 10% [36]. GN&C is often highlighted as a failure-prone subsystem and thus provides an appropriate scope of focus for the FDIR architecture. The generic subsystem taxonomy in Figure 13 shows the various high-level systems in an aerospace mission, a generic set of vehicle subsystems, and the typical components of an aerospace vehicle GN&C subsystem. Only the GN&C subsystem is expanded here because it is the focus of this study. Each of the other subsystem blocks could be expanded in a similar fashion.

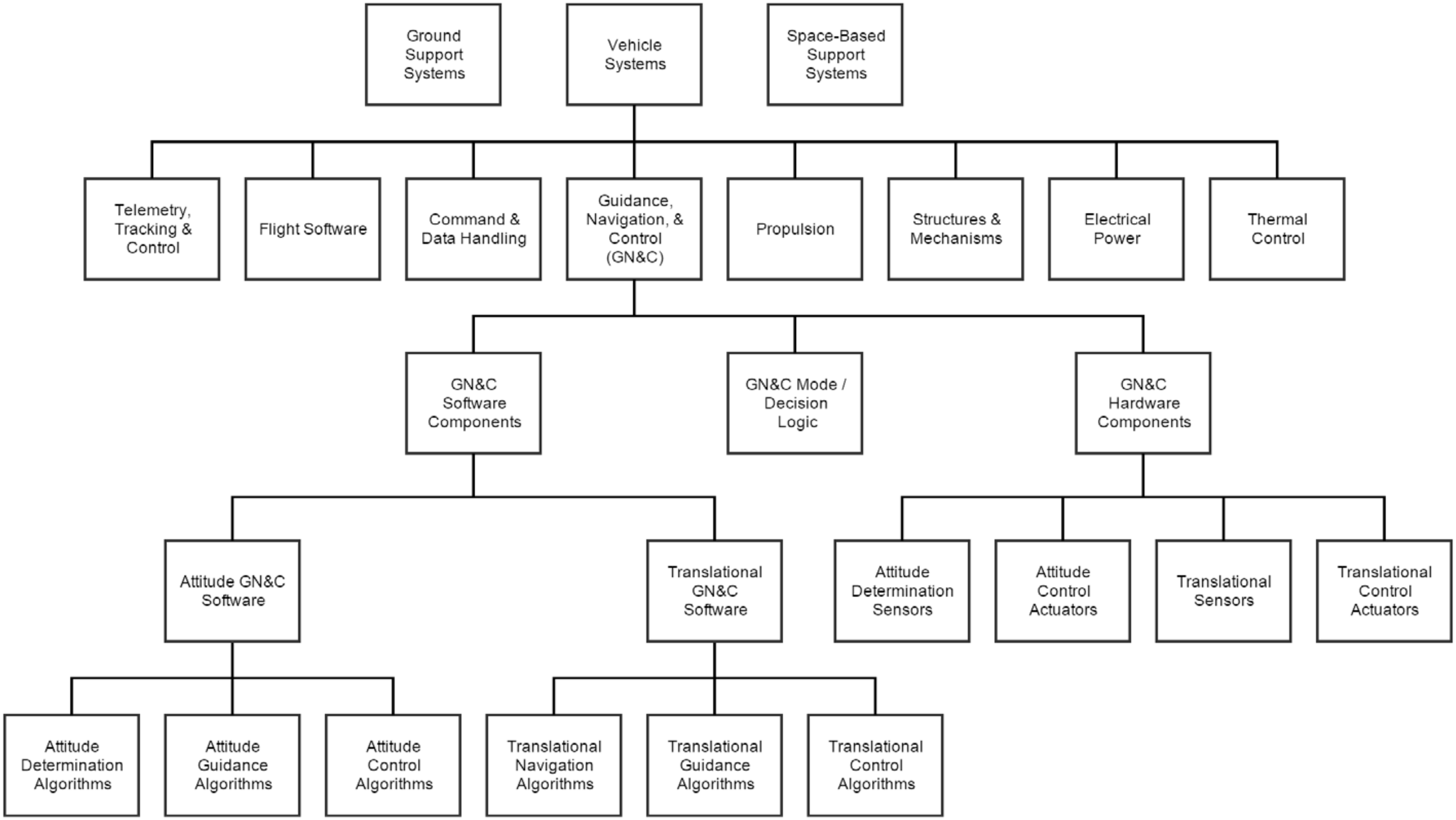

Ground Support Systems
Vehicle Systems
Space-Based Support Systems
Telemetry, Tracking & Control
Flight Software
Command & Data Handling
Guidance, Navigation, & Control (GN&C)
Propulsion
Structures & Mechanisms
Electrical Power
Thermal Control
GN&C Software Components
GN&C Mode / Decision Logic
GN&C Hardware Components
Attitude GN&C Software
Translational GN&C Software
Attitude Determination Sensors
Attitude Control Actuators
Translational Sensors
Translational Control Actuators
Attitude Determination Algorithms
Attitude Guidance Algorithms
Attitude Control Algorithms
Translational Navigation Algorithms
Translational Guidance Algorithms
Translational Control Algorithms

Figure 13 – Generic aerospace vehicle GN&C subsystem taxonomy

Almost all aerospace vehicles are reliant upon ground support systems, which range from a laptop and antenna for uploading commands to a UAV to communications ground stations and mission control centers for space missions. Many vehicles (including terrestrial ones) also depend on space-based assets such as the Global Positioning System (GPS) satellite constellation, communication relay systems, or weather satellites.

Vehicle systems include hardware and software components that are integrated into the vehicle. The GN&C subsystem is responsible for both attitude and translational motion of the vehicle. Guidance algorithms are responsible for planning where the vehicle should move or rotate. Navigation algorithms use sensors to determine the attitude state (including angular velocity and acceleration) and translational state (absolute and relative position, velocity, and acceleration) of the vehicle. Control algorithms use actuators to command the vehicle to perform both attitude and translational maneuvers. Finally, GN&C mode logic and decision software is used to manage each of these components based on the current mission phase, as well as to parse commands from ground operators.

2.3.2 Generic Fault Tree Analysis

One important quality of fault protection systems is that they should be designed in a way that all reasonably probable fault scenarios are considered and addressed. In addition to enumerating the various components that make up a system, it is necessary to determine potential sources of faults that may impact the system. It is important for mission and vehicle designers to "use historical data to determine what fault types are most likely to be introduced or … perform a risk analysis to determine what fault types would be most devastating if overlooked" [37].

One critical tool used to identify potential faults is called fault tree analysis (FTA). FTA allows system designers to identify key failure points based on the requirements and specifications of the system. An FTA provides a systematic top-down symbolic approach to model chains of possible faults for a given system [38]. The fault tree is made up of a top level event, which is a foreseeable, “undesirable event toward which all fault tree logic paths flow” [39]. Each key failure event is then traced back using conditional logical operators (such as AND and OR) to identify all possible basic fault events that could lead to that top-level failure event. The top event is connected to various intermediate events that could cause it. In turn, each intermediate event is connected to other intermediate events that could cause it. The bottom level is comprised of basic events or root cause events. These are initiating events whose cause is not analyzed further. Events are connected by AND and OR logic gates. OR implies that each one of the connected causative events is both necessary and sufficient for the resulting event to occur, even if none of the other events occur. AND implies that all connected causative events must occur and together form the necessary and sufficient condition for the resulting event. Although other logical operators are possible, most fault trees can be constructed with only the AND and OR symbols [38]. The result is a fault tree that identifies all basic fault events that should be considered by a fault protection system.

Previous literature has developed a “Fault Tree to State Machine” algorithm to transform a fault tree into a “fault state machine” that tracks the off-nominal (or “hazard”) states of a system [40]. This is distinct from a “functional state machine” that tracks all possible states of a system. The algorithm also has the capability to map states and transitions of the fault state machine to states and transitions of the functional state machine. If a functional state machine of the system is available, the fault state machine can be linked to specific system states and transitions, which are based on the hardware and software status of the system. This is preferable to the traditional rule-based approach because it allows the design of the fault state machine to be explicitly linked to the fault tree analysis and ensure that no fault conditions are missed unintentionally.

A generic fault tree analysis is shown in Figure 14. In this analysis, the top level failure is assumed to occur if any of the 2nd level events occur, as indicated by the OR gate. Each 2nd level events is in turn caused by any of the 3rd level basic events below it. For example, one of the 2nd level events, "Environmental Fault" can be caused by either 3rd level basic events or the lone 3rd level intermediate event "Physical Environment Fault", which in turn is caused by one of the 4th level basic events such as "Radiation Event" or "Debris Event". Each basic event can be mapped to a specific component of the vehicle or mission and assigned a probability if a quantitative failure analysis is desired.

2.3.3 Generic Functional State Machine

As mentioned in the previous section, a functional state machine is a model that describes the behavior of a system by tracking the "mode state" of the system [40]. Mode states are high level descriptions of the overall system behavior and are distinct from dynamic states (such as position & velocity) or vehicle component states (such as battery level or processor temperature). Each vehicle and mission will have a distinct state machine describing how these modes change and the logical conditions to switch between them. The generic functional state machine shown in Figure 15 provides a template for constructing this diagram. It features generic modes that may be present in many different

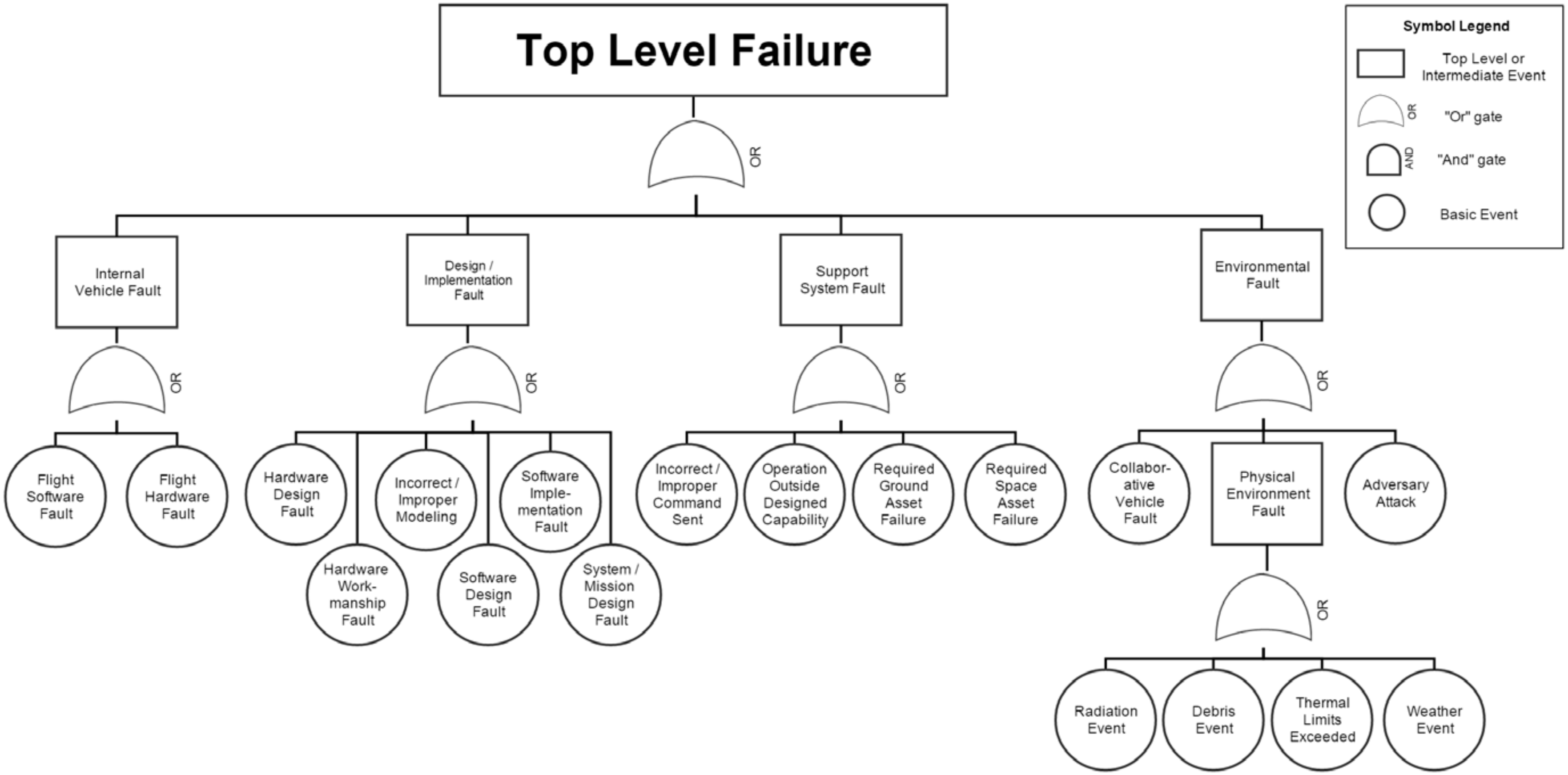

Top Level Failure
OR
Internal Vehicle Fault
OR
Flight Software Fault
Flight Hardware Fault
Design / Implementation Fault
OR
Hardware Design Fault
Incorrect / Improper Modeling
Software Imple-mentation Fault
Hardware Work-manship Fault
Software Design Fault
System / Mission Design Fault
Support System Fault
OR
Incorrect / Improper Command Sent
Operation Outside Designed Capability
Required Ground Asset Failure
Required Space Asset Failure
Environmental Fault
OR
Collabor-ative Vehicle Fault
Physical Environment Fault
Adversary Attack
OR
Radiation Event
Debris Event
Thermal Limits Exceeded
Weather Event
Symbol Legend
Top Level or Intermediate Event
OR
"Or" gate
AND
"And" gate
Basic Event

Figure 14 – Generic fault tree analysis

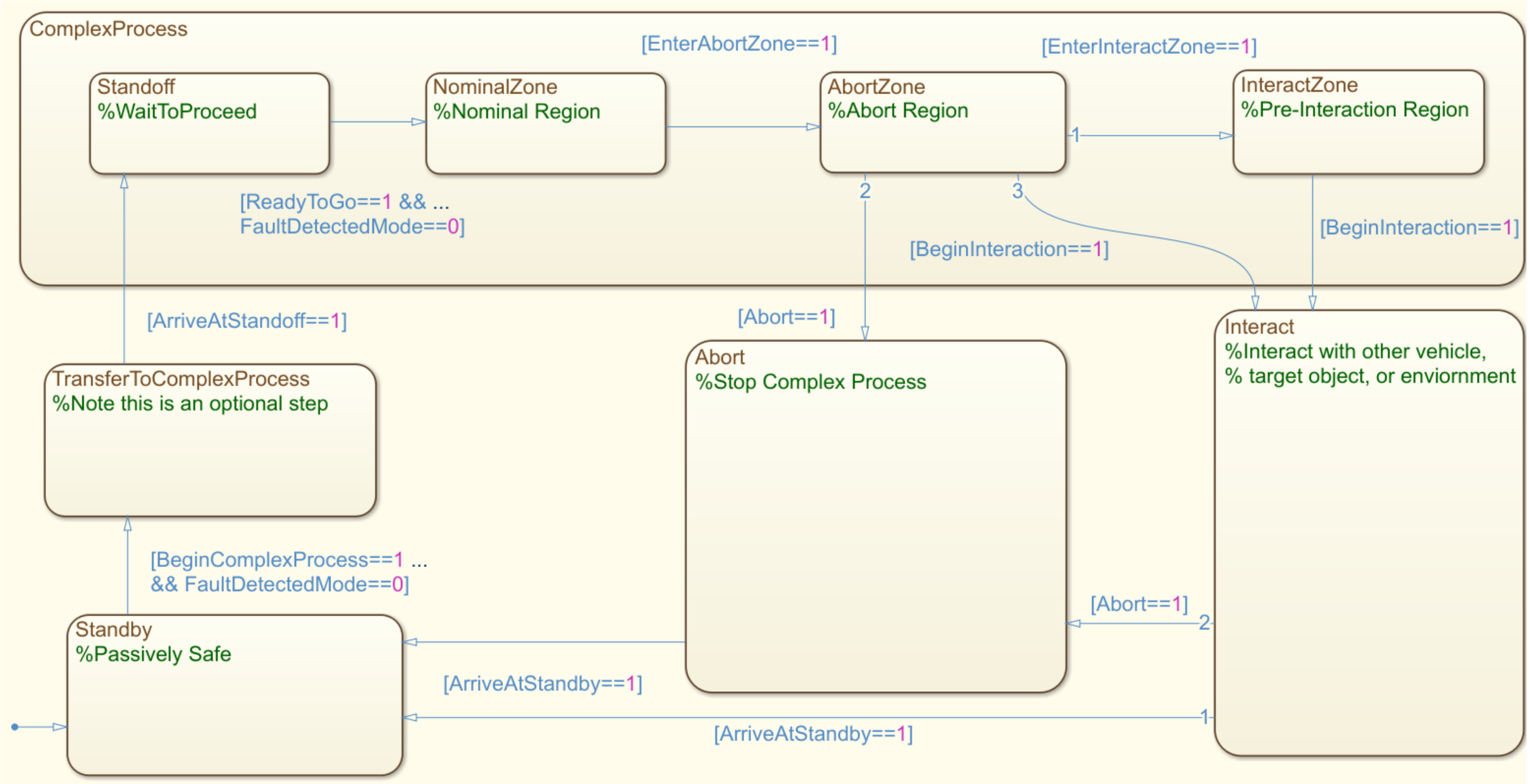


Figure 15 – Generic functional state machine

contexts. The initial state in the bottom left is “Standby,” which is a passively safe mode where the vehicle waits for further commands to proceed.

If no faults have been detected (`FaultDetectedMode=0`) then a command (`BeginComplexProcess=1`) allows a “complex process” to begin. Complex processes could include autonomous or piloted operations. An optional transition phase occurs before the complex process state begins. The complex process has several sub-states. First is “Standoff” (`ArriveAtStandoff=1`), which is a phase where the complex process is “armed” but not initiated and the vehicle is awaiting permission to proceed. Standoff is distinct from Standby because the vehicle may *not* necessarily be in a passively safe dynamic state. If no faults are detected (`FaultDetectedMode=0`) the complex process begins when a command is provided (`ReadyToGo=1`). At this point the NominalZone state begins. This is a nominal region where faults are acceptable and can generally be detected and responded to safely while still continuing nominal operations.

At some point, based on the dynamic state of the system, safe operation under fault conditions may no longer be possible (`EnterAbortZone=1`). When this occurs, the AbortZone state begins, and at any time if a fault detection is triggered or a human operator decides conditions are unsafe, an Abort can be commanded (`Abort=1`). The abort stops the complex process and moves the vehicle to a safe dynamic state, eventually returning to the Standby state (`ArriveAtStandby=1`). Additionally, an “Interact” state allows the vehicle to interact with other vehicles, target objects, or the environment. A pre-interaction region called the InteractZone is entered from the AbortZone when (`EnterInteractZone=1`). The Interact state can be entered from either AbortZone or InteractZone when a command is received (`BeginInteraction=1`). The vehicle *cannot* enter the Interact state directly from NominalZone because interaction almost always involves hazardous conditions. If a fault or other hazard occurs during Interact, an abort can be triggered (`Abort=1`). If no anomalies occur, the vehicle will return to passively safe standby after the interaction is complete (`ArriveAtStandby=1`).

2.3.4 Generic Diagnostic State Machine

To implement on-board model-based fault diagnosis, the generic diagnostic state machine shown in Figure 16 has been developed. The diagnostic state machine consists of two primary states: NoFaultDetected and FaultDetected. During all nominal mission phases, NoFaultDetected is activated, but when a fault detection trigger is observed (`FaultDetected=1`), the FaultDetected state will be activated. If the functional state machine is in any state other than AbortZone, then the diagnostic state machine will enter "Diagnose" immediately when a fault is detected. If the functional state machine is in the Abort Region (`AbortZone=1`) when a fault is detected, the diagnostic state machine does not attempt to determine which fault has occurred. An abort maneuver is commanded immediately, returning the vehicle to a passively safe dynamic condition before entering the Diagnose state.

The Diagnose state consists of sub-states for each possible fault. Each sub-state begins by running a diagnostic routine to determine if that particular fault has occurred. If the diagnostic routine returns `FaultConfirmed=1`, then the appropriate fault response routine is called and the diagnostic sub-state for the next fault is activated while the response runs in the background. If the diagnosis does not result in fault confirmation within a user-defined wait time, then the active sub-state moves to the next possible fault

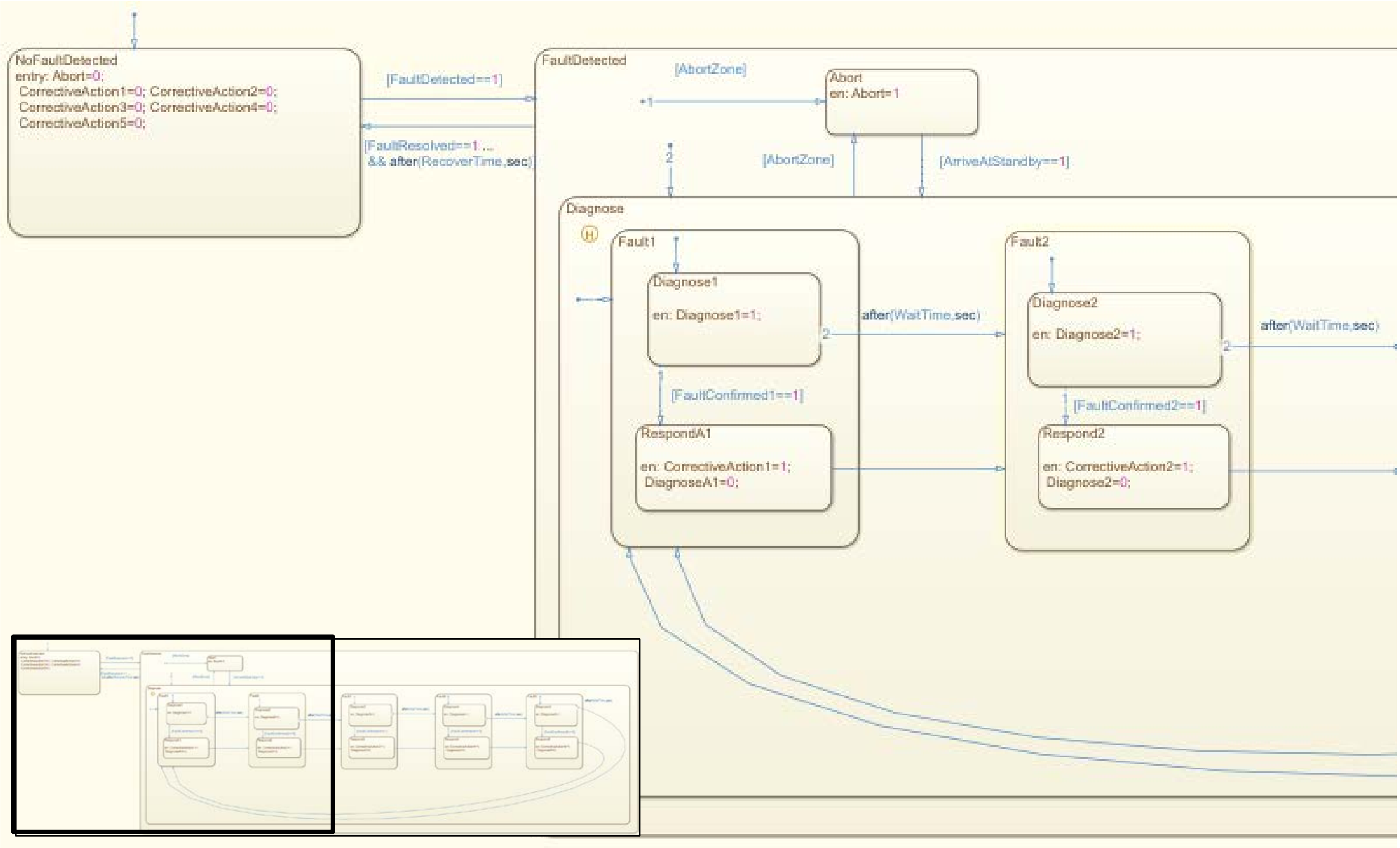


Figure 16 – Generic diagnostic state machine

and the process repeats. Once all possible faults have been evaluated, the active sub-state returns to the first fault until the fault has been resolved by one of the corrective actions.

Note that fault *diagnostic* checks are distinct from fault *detection* checks. None of the diagnostic checks are performed unless they are called by the diagnostic state machine, which is only activated once the fault detection triggers are activated. Thus, a fault will *not* be detected if one of the fault *diagnosis* conditions is met but the fault *detection* conditions have not been met. Once a fault has been diagnosed, the diagnostic state machine calls the appropriate fault response routine. When the fault is resolved and a user-specified recovery time has passed, the active state returns to the NoFaultDetected state.

2.3.5 Generic Architecture Block Diagram

The functional state machine and diagnostic state machine described in the previous two sections are designed to work together in the fault protection architecture along with several additional components in the MATLAB/Simulink environment. An overview showing the architecture components and their connections is illustrated by the block diagram in Figure 17. The diagram is divided into the system under control (entity being controlled) and the control system (entity exercising the control) according to the terminology defined in [5].

During initial development and testing, the system under control is composed of simulation models and state variables that track the status of these models. For example, environment models keep track of time systems, dynamic perturbations on the vehicle, and ephemerides for the positions of planets, moons, and the Sun. Environment states include

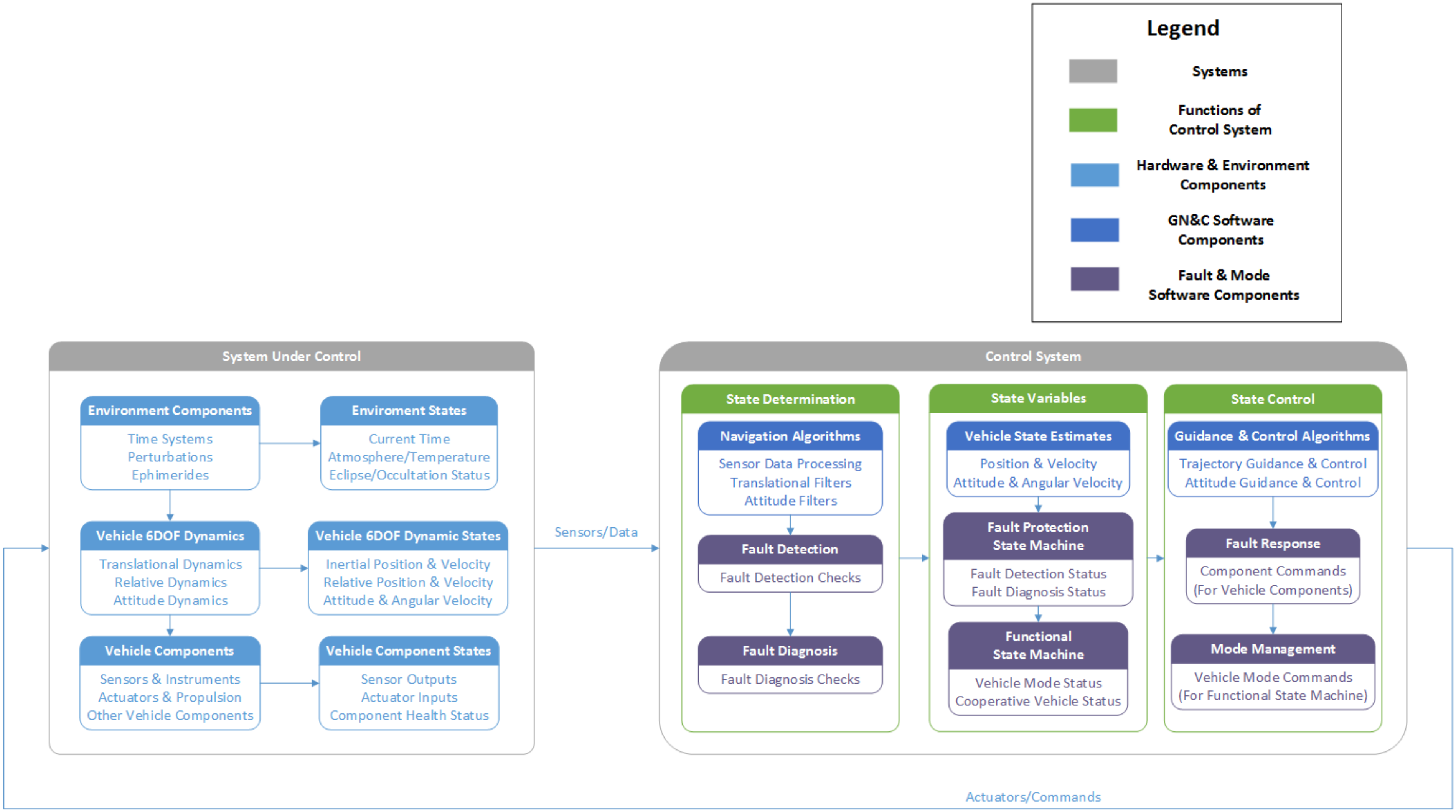


Figure 17 – Generic architecture block diagram

the current time, atmospheric and temperature conditions, and eclipse/occultation status for planets and moons. The simulation environment also includes vehicle dynamic models for inertial and relative translational dynamics (position, velocity, acceleration) and inertial attitude dynamics, including angular velocity and acceleration. Generic vehicle sensor/instruments and actuator/propulsion models are also included in simulation to provide sensor outputs and receive actuator inputs based on the state of the simulation. Other vehicle components and their health status can also be accounted for. Again, these simulation models are only a development tool and are not used as part of the on-board software system for the FDIR architecture when it is deployed on a flight computer.

The “Control System” contains the core of the FDIR architecture as well as vehicle GN&C algorithms and state estimates. The GN&C algorithms include sensor data processing and filters for estimating vehicle position, velocity, attitude, and angular velocity and guidance & control algorithms for both vehicle trajectory and vehicle attitude. The fault & mode portions of the architecture are the main focus of this study, and these components are shown in the example Simulink diagram in Figure 18, which illustrates how each of the components interacts with the others.

The two primary components are the functional state machine and the diagnostic state machine. These are Stateflow blocks which have been described in the previous two sections. Most of the inputs to the functional state machine are produced by the generic mode management block, a MATLAB function which takes in vehicle state information and ground commands and calculates the logical variables that are evaluated in functional state machine transitions. One output from the functional state machine to the diagnostic state

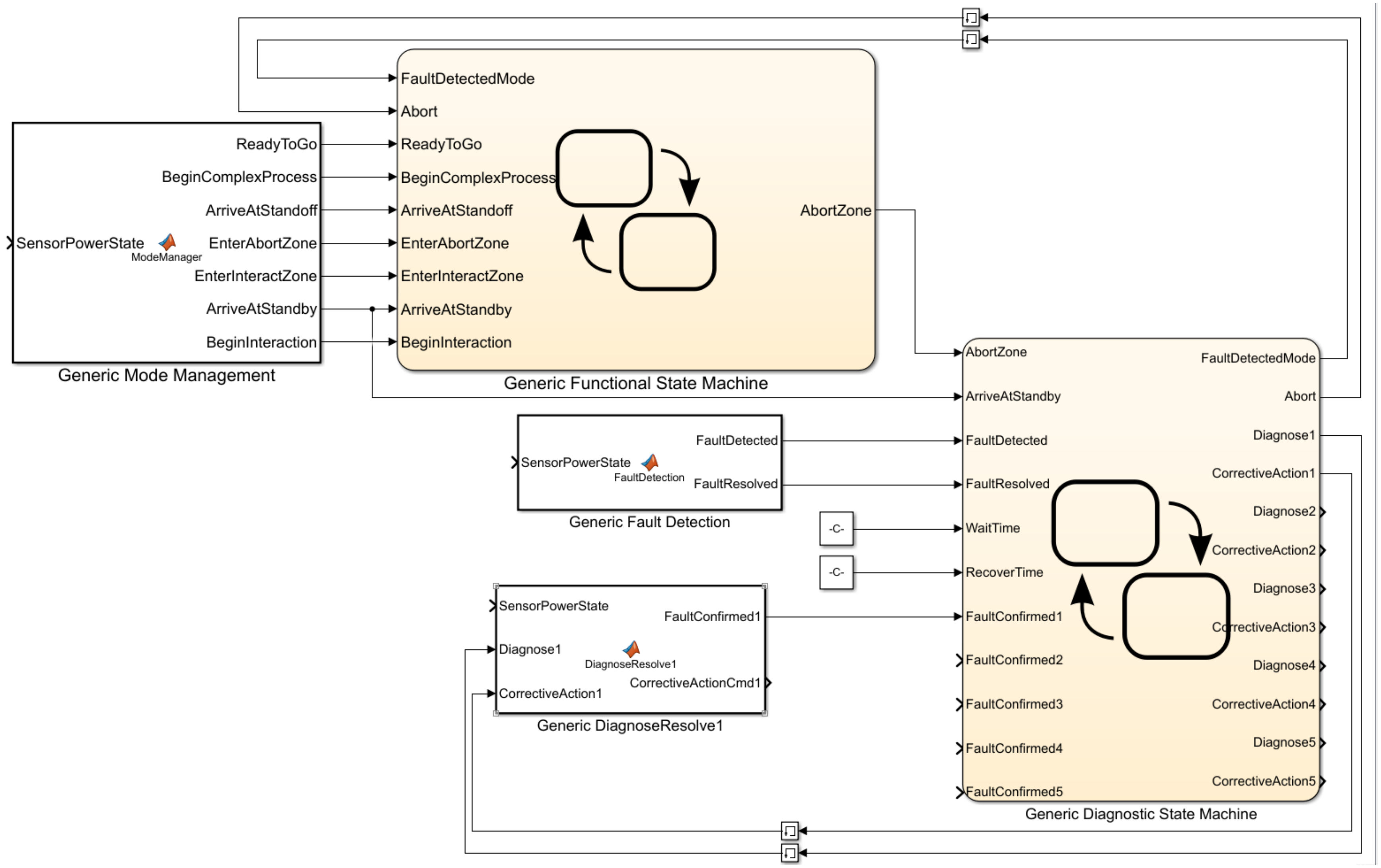


Figure 18 – Simulink diagram for generic fault protection architecture

machine (`AbortZone`) describes whether the AbortZone state is active. Two inputs (`FaultDetectedMode` and `Abort`) are generated by the diagnostic state machine.

The inputs to the diagnostic state machine come from several sources. One variable (`ArriveAtStandby`) is generated by the mode management MATLAB function block. Another variable (`AbortZone`) is generated by the functional state machine. Fault detection checks are performed by a MATLAB function block and result in two variables (`FaultDetected` and `FaultResolved`). Two variables are input as constants (`WaitTime` and `RecoverTime`), and a set of variables indicating fault confirmation (`FaultConfirmed1,2,3,` etc.) are input from the fault diagnosis/resolution MATLAB function block for each fault. Two output commands for each fault (`Diagnose1,2,3,` etc. and `CorrectiveAction1,2,3,` etc.) are fed into the respective fault diagnosis/resolution function blocks. Note that only one diagnosis/resolution function is shown for clarity but most systems will consider more than one fault and will have a diagnosis/resolution function for each fault.

2.4 Mapping the Generic Architecture to a Particular Application

Each of the generic architecture diagrams described in the preceding section can be adapted for particular applications. This section presents a general process for mapping from generic to specific, and the following two chapters provide examples for two very different applications. It is important to note that the generic diagrams provide more or less detail than necessary, depending on the application. Detail can be added or removed in each diagram as needed.

The first step in adaptation is to adjust the generic aerospace vehicle subsystem taxonomy in Figure 13 to the specific vehicle being considered. The goal is not to provide scrupulous detail of every miniscule component but instead to identify and categorize the main components that may have an impact on tasks and processes relevant to fault protection. The engineer must use their judgement to determine the level of granularity required, but in general a component-level taxonomy should be sufficient. Subsystems and GN&C components can be added or removed from the generic taxonomy as needed. Although this diagram is not used directly to generate other aspects of the overall architecture, it provides an extremely useful reference for standardizing terminology and vehicle configuration.

The next step involves performing a fault tree analysis for the application. The template in the generic fault tree in Figure 14 can be used as a guide for this, or the fault tree structure can be followed without using the same intermediate and root cause events. Additionally, other industry standard analyses such as the Failure Modes, Effects, and Criticality Analysis (FMECA) or Fault Containment Regions (FCR) can be developed [12]. An important step in this process is determining which faults are considered credible and must be addressed by a FP system and which faults can be ignored as accepted risks. Quantitative risk metrics are useful for this purpose but are beyond the scope of this study.

Next, a functional state machine should be created, using the generic functional state machine in Figure 15 as a template. This can be done in Stateflow, a state machine toolbox within MATLAB/Simulink, or in alternate software tool such as UML/MagicDraw. Some states from the generic diagram such as TransferToComplexProcess, Abort, and Interact may not be relevant for all applications, and some applications may require that additional states be added. Transition conditions between states can also be modified, added, or removed as needed. The main purpose of this diagram is for other aspects of the FDIR architecture such as the diagnostic state machine to reference the mode state of the vehicle. Additionally, supporting functions such as the mode management MATLAB block should be written to generate the logical inputs used in transitions between states. Although alternate representations of mode state (such as simple code functions or Simulink blocks) can also be used, the state machine paradigm provides a useful graphical tool that should be relatively straightforward to interpret without intimate knowledge of the system.

A diagnostic state machine should also be developed from the template in Figure 16 using selected faults from the fault tree. The behavior of fault responses during the AbortZone should be determined, as this is reflected in the diagnostic state machine. Also, supporting functions for fault detection, diagnosis, and response should be written to provide inputs to and receive commands from the diagnostic state machine. These functions can be placed in MATLAB code blocks and connected to the functional and diagnostic state machines as shown in Figure 18.

Finally, a V&V method such as a numerical simulation, testbed, or flight test should be should be developed to evaluate the FP architecture using the template shown in the block diagram in Figure 17. The block diagram should be adapted to the particular application as a reference for understanding how all components interact. Creating this diagram also helps the developer to determine if any important components have been left out. Once the FP architecture has been thoroughly evaluated, it is ready to be deployed. The following two chapters provide examples of this process for both a terrestrial UAV application and a space-based automated proximity operations application.

CHAPTER 3. TERRESTRIAL APPLICATION: FALCONVIZ UAV NERVOUS SYSTEM

One application of the state machine FDIR architecture has been developed for a multirotor UAV system. This has been dubbed the “UAV Nervous System,” and serves as a proof of concept of the state machine FDIR architecture. Several flight tests demonstrating successful detection of faults have been completed. The results of one set of flight tests were published at the 67th International Astronautical Congress in Guadalajara, Mexico [41].

3.1 Overview of FalconViz UAV Hardware and Typical Missions

FalconViz is a start-up company based out of the King Abdullah University of Science and Technology (KAUST) in Saudi Arabia. It was founded in 2015 by two research faculty and a PhD student at KAUST: Dr. Neil Smith, Dr. Mohamad Shalaby, and Luca Passone. FalconViz designs and flies custom UAVs for a variety of applications such as aerial surveying & mapping, inspection & monitoring, and surveillance. The company also collaborates with other research groups at KAUST such as the Hydrology, Agriculture and Land Observation (HALO) group led by Dr. Matthew McCabe. The HALO group uses modeling, remote sensing, and in-situ measurements to better understand elements such as water usage, crop health, and regional climate conditions. One effort of the HALO group involves the use of UAVs to capture thermal and hyperspectral imagery of desert agricultural plots.

At KAUST during Summer 2016, a FalconViz hexacopter (six rotors) shown in Figure 19 was used as a proof-of-concept testbed for the UAV Nervous System. One specific fault was addressed as a starting point: unbalanced propellers (leading to excess vibration). Detecting this fault provides a more reliable vehicle for performing aerial surveys and other tasks with FalconViz UAVs. Additional modifications and flight tests were performed during Summer 2017 to add more vibration sensors and the capability to detect additional faults using temperature, voltage, and current sensors.

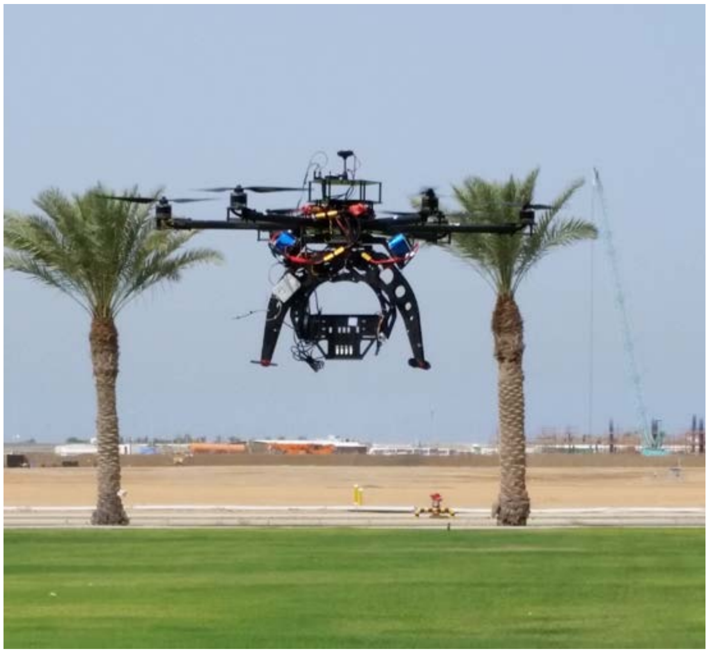


Figure 19 – FalconViz hexacopter in flight [41]

3.2 UAV Nervous System Requirements

Unbalanced propellers in multi-rotor UAVs cause excess vibration and can lead to screws coming loose and potential crashes. The basic requirement for the first iteration of the UAV Nervous System is to detect excess vibrations and provide a response action. This must be accomplished in real-time via onboard measurements only and should utilize the Stateflow FDIR architecture developed in MATLAB/Simulink. Also, the detection should not be triggered unless an unbalanced propeller is confirmed. In other words, intermittent fault triggers are not desired; the fault status should not constantly flip back and forth between “fault” and “no fault” during flight. In addition, the UAV Nervous System should be a standalone system that has minimal impact on the operation of the UAV. It should not add excessive mass or power drain to the UAV.

3.3 FalconViz UAV Subsystem Taxonomy and Fault Tree

The generic subsystem taxonomy has been adapted for the FalconViz UAV as shown in Figure 20. Most FalconViz UAV components are Commercial Off-the-Shelf, while the structure and overall design of the UAVs are produced in-house. A few of the fault-prone components are outside of the GN&C subsystem but directly affect it. For example, batteries and Electronic Speed Controls (ESCs) are components of the EPS that provide power to GN&C components such as propeller motors. The telemetry receiver, which connects the UAVs to hand controllers and ground stations, is a key TT&C component. Key GN&C components include the six motors and propellers, which are the sole attitude and translational control actuators on the vehicle. Note that the propellers of a multirotor UAV typically have fixed pitch; they perform both attitude and translational maneuvers by altering the speed of individual propeller rotations via the ESCs to change the direction of the resultant lift/thrust vector [42]. Some key translational sensors include a GPS receiver, a barometer, and accelerometers. Key attitude sensors are a rate gyroscope and a magnetic compass (magnetometer). Most GN&C software is embedded within the flight controller, including a set of translational guidance waypoints that can be uploaded by mission planning software on a ground station laptop. The GPS satellite constellation is also used by the GPS receiver to calculate the UAV’s position and velocity.

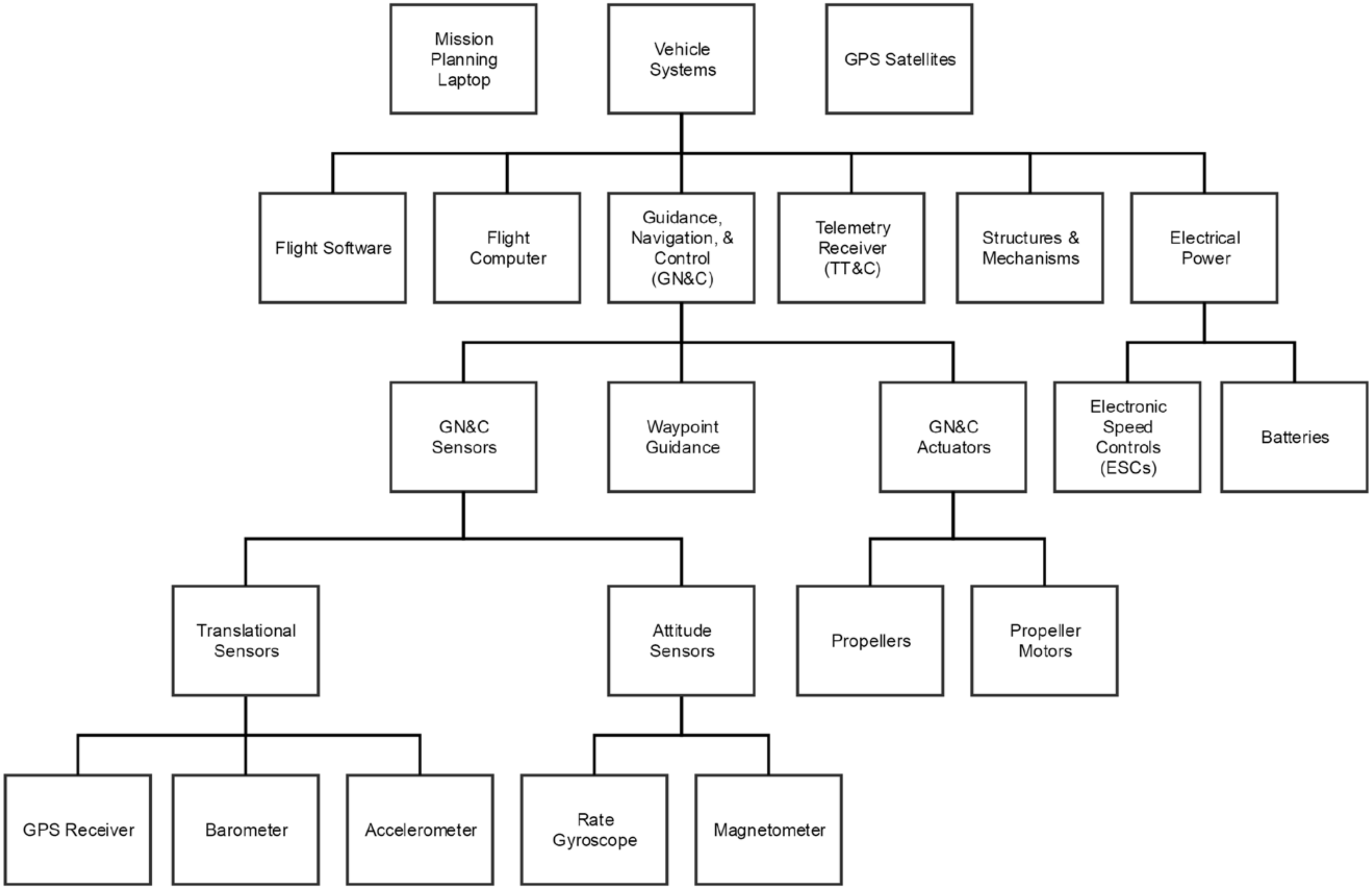


Figure 20 – Subsystem Taxonomy for UAV Nervous System

A fault tree for the FalconViz UAVs is shown in Figure 21. This is an expansion of the Internal Vehicle Flight Hardware branch of the generic fault tree from Section 2.3.2 and shows basic event faults identified by the FalconViz engineering team. The top-level failure for this analysis is “Loss of control” which can be traced back to each the identified faults. The three most important faults identified were “Motor not spinning,” “Motor or ESC overheats” and “Excessive vibration.”

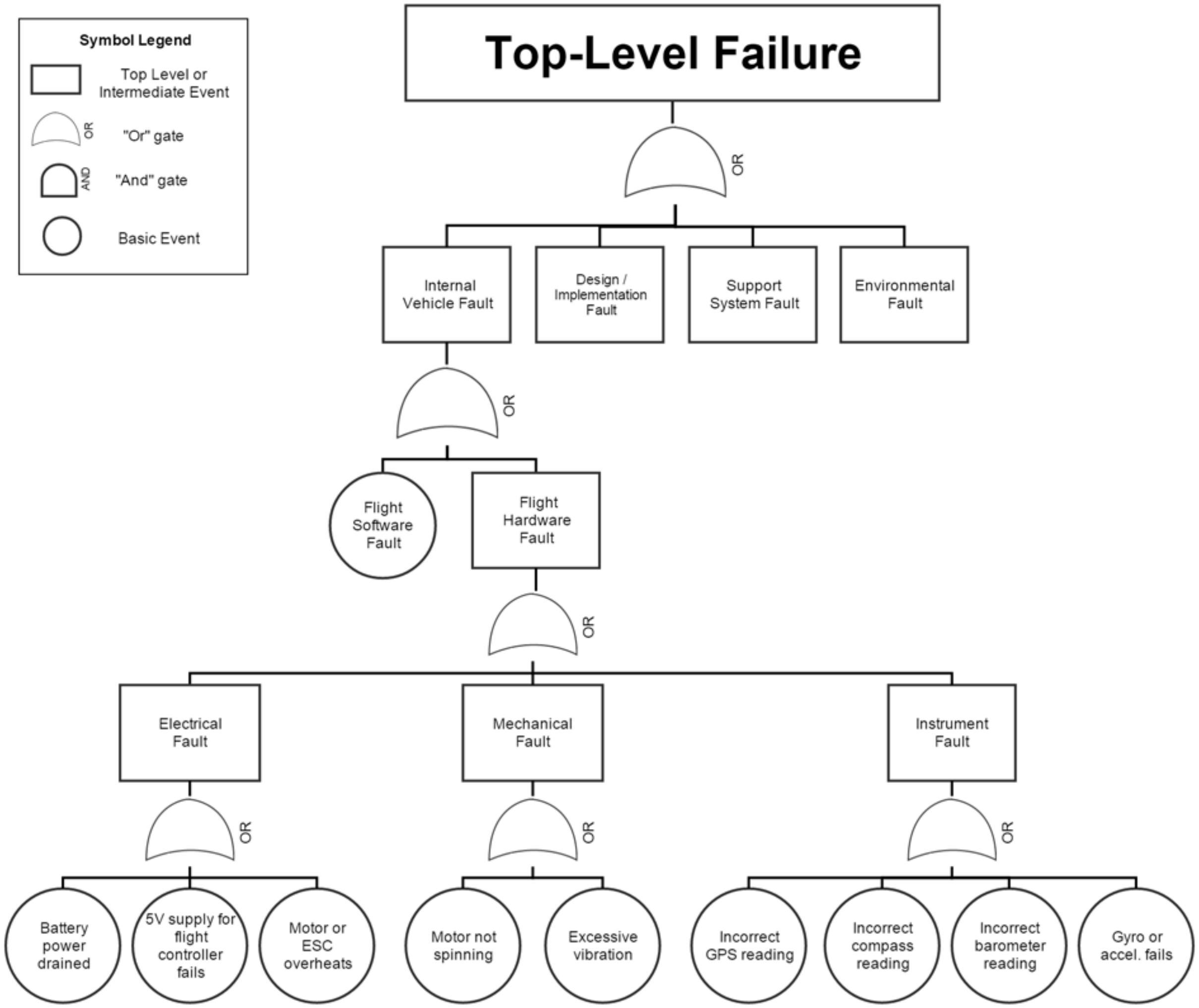


Figure 21 – Fault Tree for FalconViz UAVs

3.4 Fault Detection and Recovery Strategy

Vibration detection is accomplished by evaluating accelerometer data measured from the arms of the UAV that house the propellers. A machine learning algorithm determines the health of the system from the data. If the propellers are unbalanced, then there will be much more vibration in the system. Once it is trained and validated on the ground, the machine learning model then identifies the health of the system from live data onboard the UAV. These outputs are sent into the state-based FDIR architecture in Stateflow.

A SparkFun Triple Axis Accelerometer and Gyro Breakout – MPU-6050 [43], shown in Figure 22a, is installed on one arm of the hexacopter. Data is collected via a microcontroller programmed with Arduino protocols called the Teensy 3.2 [44], shown in Figure 22b. The Teensy is then connected to a MeegoPad T02 compute stick [45], shown in Figure 22c, via USB.

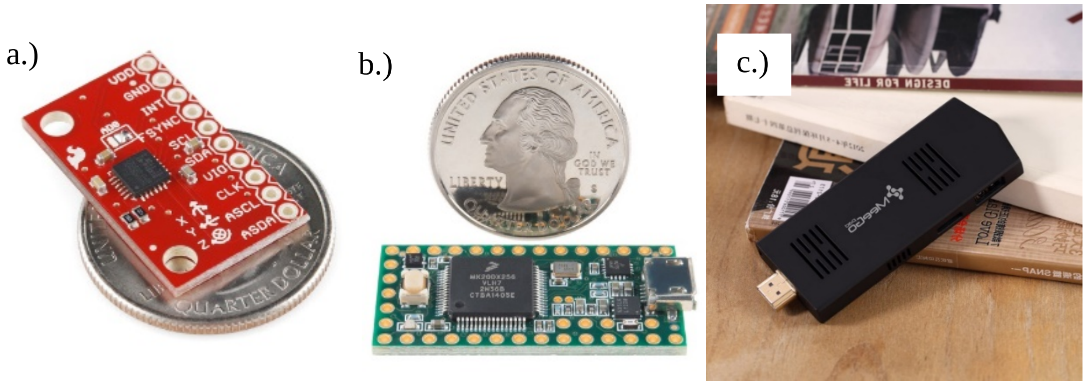


Figure 22 – (a) SparkFun Triple Axis Accelerometer & Gyro Breakout – MPU-6050 [43]; (b) Teensy 3.2 [44]; (c) MeegoPad compute stick [45]

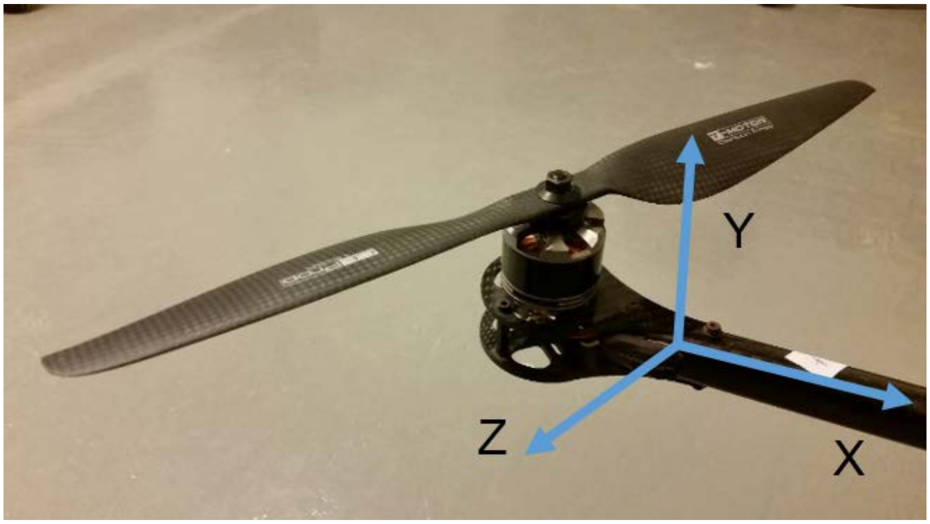


Figure 23 – Shrink-wrapped accelerometer installed on hexacopter arm, with sensor coordinate axes indicated [41]

The shrink-wrapped MPU-6050 breakout board is mounted just below the propeller motor, as shown in Figure 23. The Teensy is installed on a SparkFun Teensy Arduino Shield Adapter [46] and connected to the MPU-6050 and other components via jumper cables and custom harnesses, as shown in Figure 24.

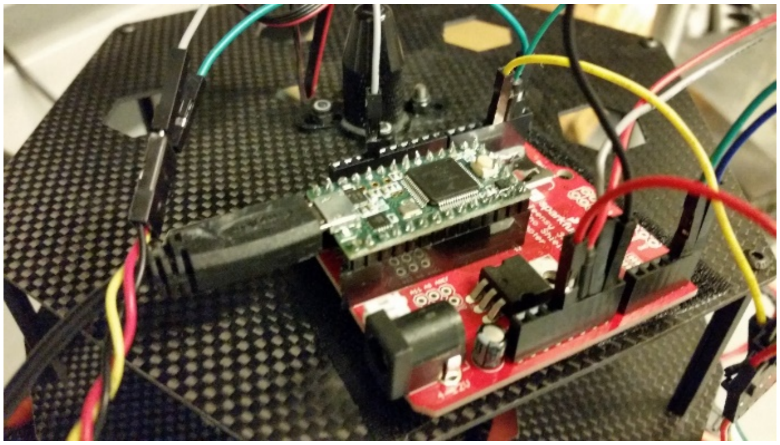


Figure 24 – Teensy installed on Arduino Shield Adapter [46] with USB and jumper cable connections [41]

If the propellers are unbalanced, there will be much more vibration in the system. A Simulink model run in Windows on the MeegoPad records data from the accelerometer and feeds it through a supervised machine-learning classification algorithm in MATLAB called K-nearest neighbors (KNN) [47]. The KNN algorithm determines the health of the system from the data. Once it is trained and validated on the ground, the KNN model then identifies the health of the system from live data onboard the UAV. These outputs are sent into the state-based FDIR architecture in Stateflow. For tests, the propeller is unbalanced by adding a few pieces of electrical tape on one side, as shown in Figure 25.

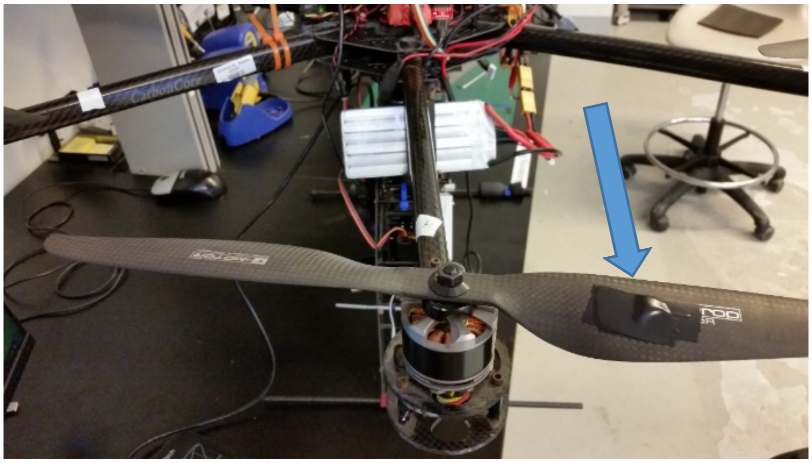


Figure 25 – Hexacopter propeller unbalanced by adding electrical tape [41]

Flight test data is captured for both an unbalanced propeller (with tape) and a balanced propeller (without tape) and is used to train the KNN classification model in MATLAB on the ground. Training data plotted in Figure 26 shows that the unbalanced propeller has a higher vibration magnitude in y and z and that the y data shifts into the negative region. However, this information is not provided to the KNN algorithm.

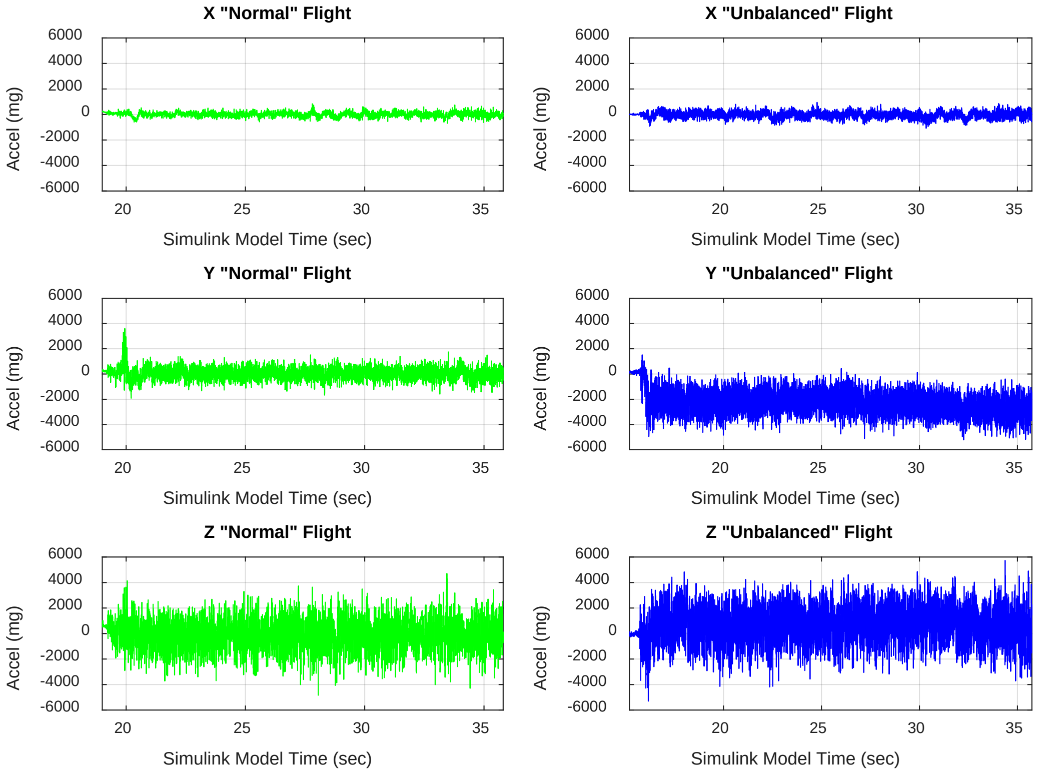


Figure 26 – Training data for KNN classification model [41]

Raw data from the two flights is combined and manually labelled by the user. Combined raw data for each axis (x,y,z) and assigned labels are fed into the KNN training algorithm in MATLAB. After training, the static KNN model is stored for use in flight and is not adapted further. The detection accuracy of the static model is verified with independent flight test validation data captured in the same way as the training data. The KNN classification detection algorithm then uses the trained KNN model to select the labels for each data point. The validation results, shown in Figure 27, exhibit a detection accuracy of 86.8%.

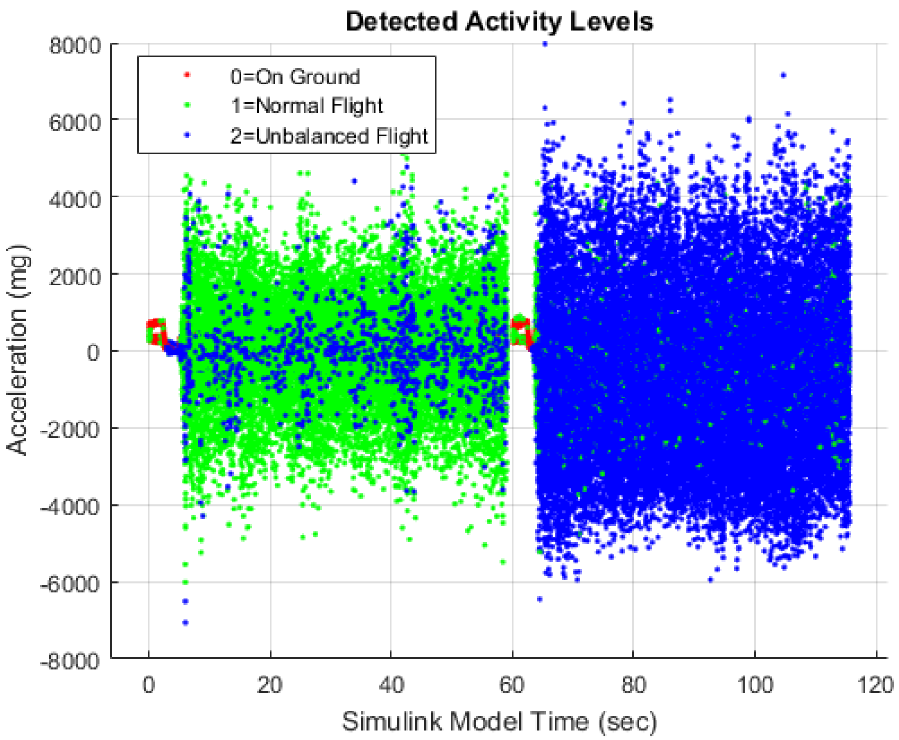


Figure 27 – Detected activity levels from KNN model validation data [41]

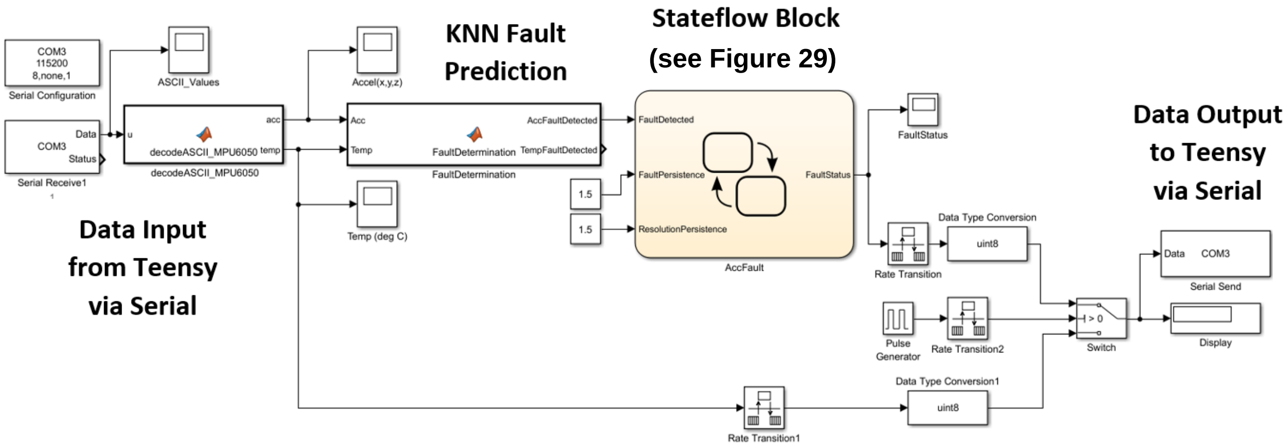


Figure 28 – Simulink diagram for UAV nervous system [41]

Once model training and validation is complete, the system is ready for in-flight detection. Data collected from the accelerometer in real-time is fed into the Simulink model shown in Figure 28 via a serial connection over USB. The Teensy and serial connections run at 115,200 baud (bits per second). The data is converted from ASCII characters to numerical values by a custom MATLAB function and is saved to memory. It is then fed into the MATLAB KNN fault detection algorithm, which uses the trained static model to determine if the propeller is balanced or not. The detection is run on 100 samples at a time, and if 50 or more of these samples are classified as "unbalanced" by KNN, then the vibration `FaultDetected` flag is set to 1; otherwise the flag is set to 0, indicating the propeller is "balanced". This `FaultDetected` flag is fed into the Stateflow diagram shown in Figure 29, which conducts state machine fault protection. Note that this diagram was developed before the generic architecture discussed in the previous chapter. It is adapted to fit the generic architecture in the Section 3.7.

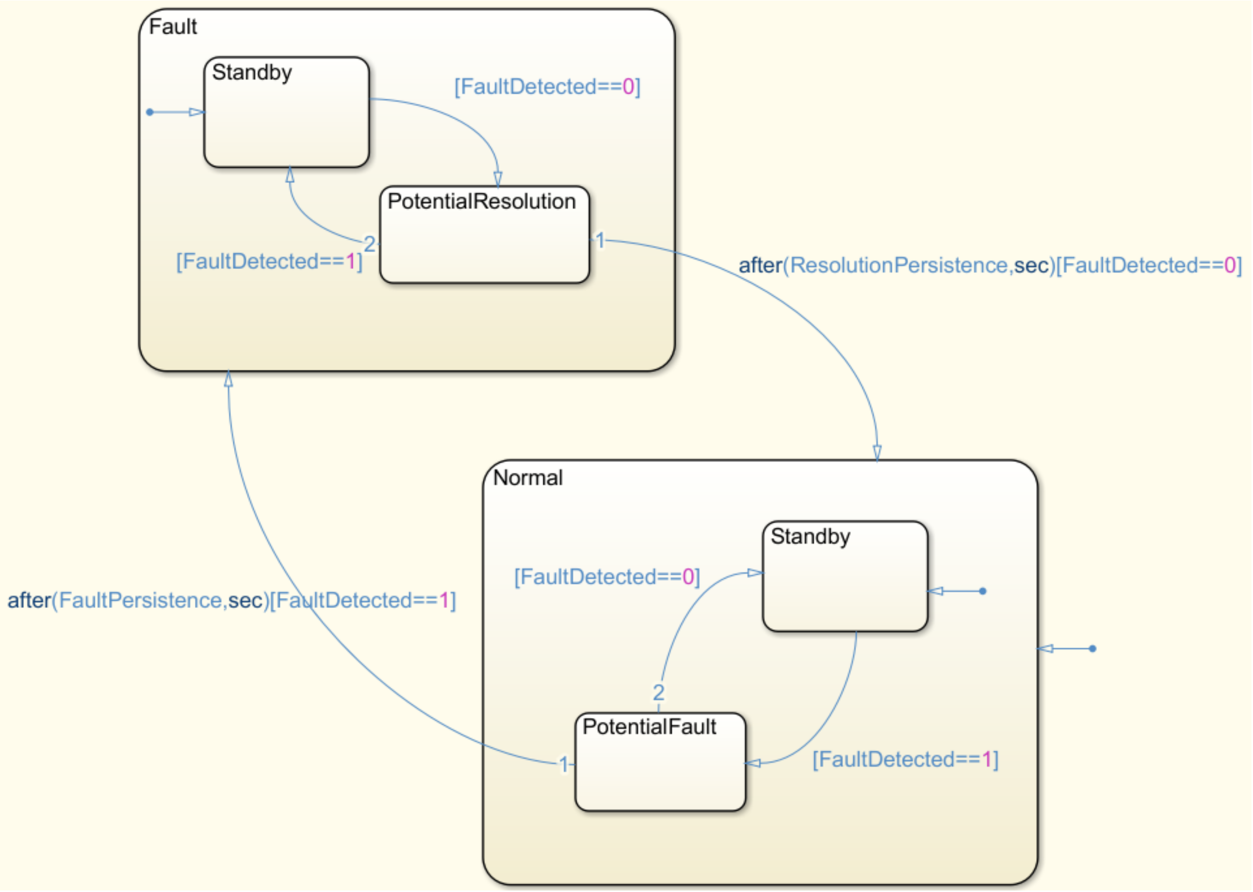


Figure 29 – Stateflow diagram for vibration fault detection [41]

The Stateflow diagram begins with an initial state of "Normal" at the bottom right and an initial substate of "Standby". If `FaultDetected` is set to 1, the substate within

“Normal” transitions to “PotentialFault.” If the condition “`FaultDetected=1`” persists for a length of time specified by `FaultPersistence`, then the state transitions from “Normal” to “Fault”. However, if `FaultDetected` does not remain at 1 for long enough, then the state will remain “Normal” and the substate will return to “Standby”. Very similar logic applies for transitioning from “Fault” back to “Normal”: the condition `FaultDetected=0` must persist for a length of time specified by `ResolutionPersistence`. The `FaultStatus` flag is the output signal matching the current state of the Stateflow chart, with 0 indicating “Normal” and 1 indicating “Fault”. The fault response involves sending the `FaultStatus` signal to the telemetry receiver. Simulink sends the `FaultStatus` signal back to the Teensy and then on to the FrSky X8R telemetry receiver [48] shown in Figure 30a. The pilot can view the value of `FaultStatus` on their Taranis X9D handheld radio controller [49], shown in Figure 30b, to indicate whether the propeller is balanced or not (0 or 1). If the variable is set to 1, the controller is programmed to begin beeping. When the variable is set to 0, the controller stops beeping.

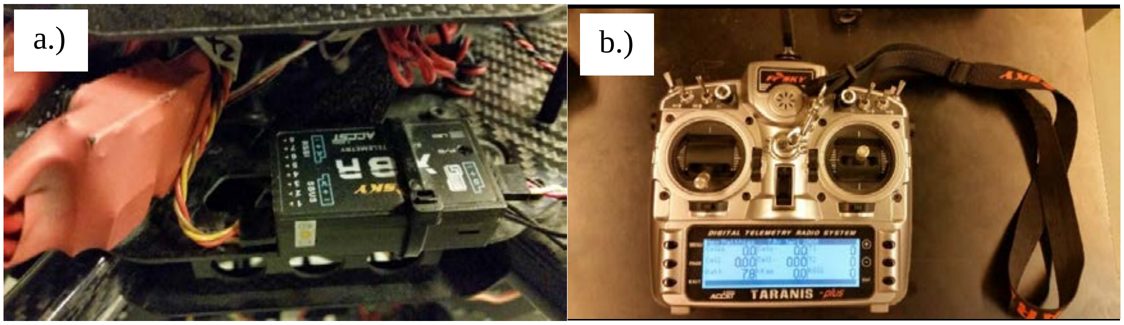


Figure 30 – a.) FrSky X8R telemetry receiver [48] installed on the hexacopter;
b.) Taranis X9D Plus radio [49]

3.5 Flight Test Results

The UAV Nervous System has been flight-tested and successfully indicates the state of vibrations during flight. Figure 31 shows data recorded during the V&V test flight. The top three plots show accelerometer data and the bottom plot shows the `FaultStatus` signal output by the Stateflow diagram. The flight begins with the copter on the ground in segment A, and tape is placed on the propeller to unbalance it. The copter takes off and flies with an unbalanced propeller in segment B. The nervous system quickly detects the imbalance and outputs a `FaultStatus` of 1 at around 12 sec, shortly after segment B begins. During segment C, the copter lands, and the tape is removed to restore the propeller balance. Segment D shows balanced flight, and around 25 sec the nervous system detects that balance has been restored and sets `FaultStatus` to 0. The copter lands again in segment E and tape is added again. During segment E near 35 sec, a `FaultStatus` of 1 occurs, and since there is no persistent “normal flight” data entering the system, `FaultStatus` does not return to 0. Unbalanced flight resumes during segment F, and `FaultStatus` remains at 1. The tape is not well adhered to the propeller during this segment, and it comes loose and flies off at 40 sec. The copter transitions to balanced flight in segment G, which the nervous system detects around 45 sec, returning `FaultStatus` to 0. The delays in `FaultStatus` transitions are expected, as the system is tuned to avoid constant flipping between 0 and 1.

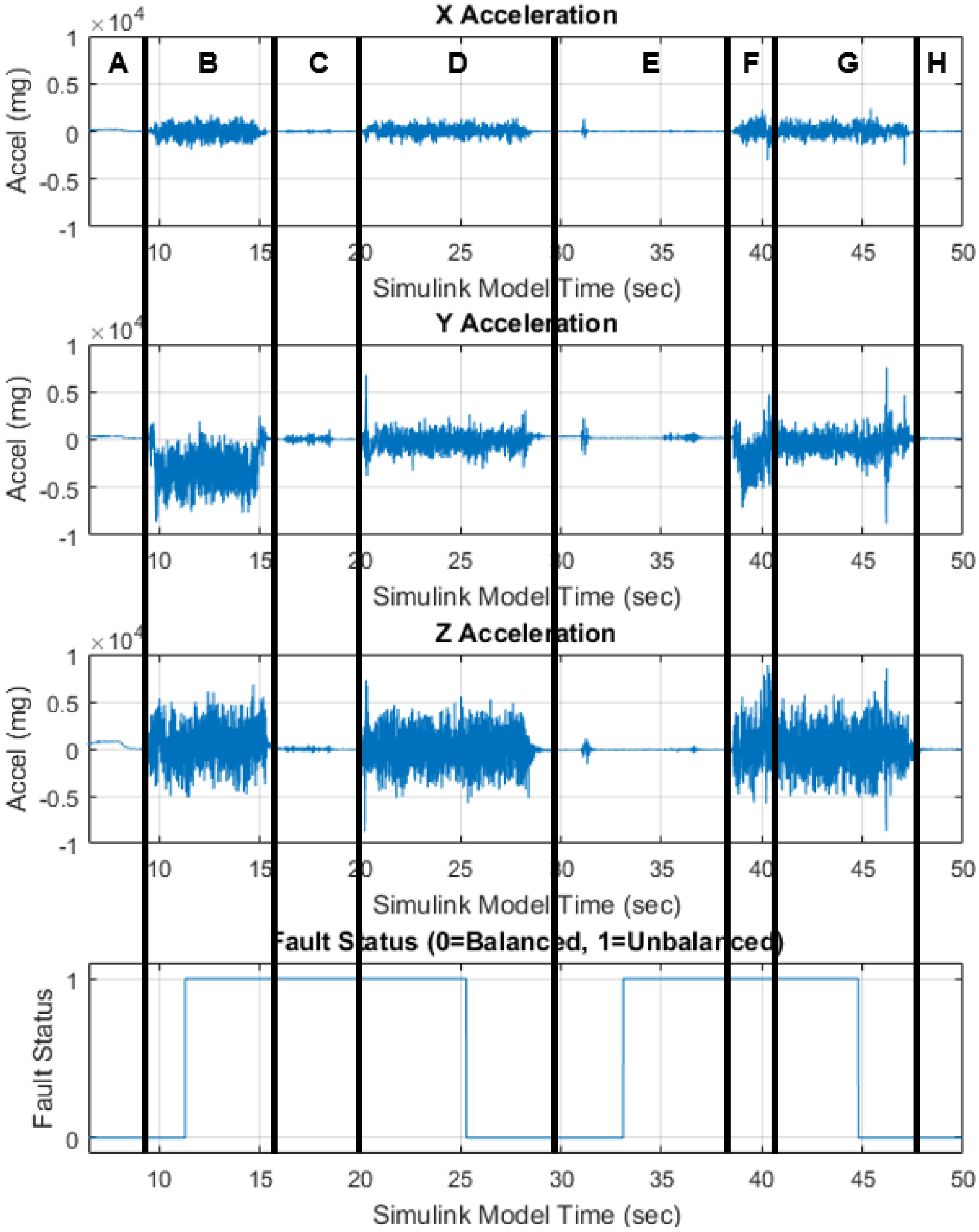


Figure 31 – Flight test data demonstrating successful vibration detection [41]

3.6 UAV Nervous System Upgrades

The flight test described above provided a proof-of-concept for the UAV Nervous System. In addition to the vibration sensor, a One Wire Digital Temperature Sensor DS18B20 [50] is used to monitor heating of the motors. It is important to detect when motors overheat because their shutdown and can lead to loss of the vehicle. Figure 32a shows the DS18B20 sensor and Figure 32b shows it installed on a motor with thermally conductive epoxy. The temperature reading in deg C is collected by the Teensy then downlinked to the Taranis radio via the telemetry receiver. This value is displayed on the radio for the pilot, and the radio is programmed to give a verbal warning (“too high”) when the temperature exceeds a predetermined threshold. This threshold is set by the pilot on the radio itself. An example plot of saved temperature values from the DS18B20 is shown in Figure 33.

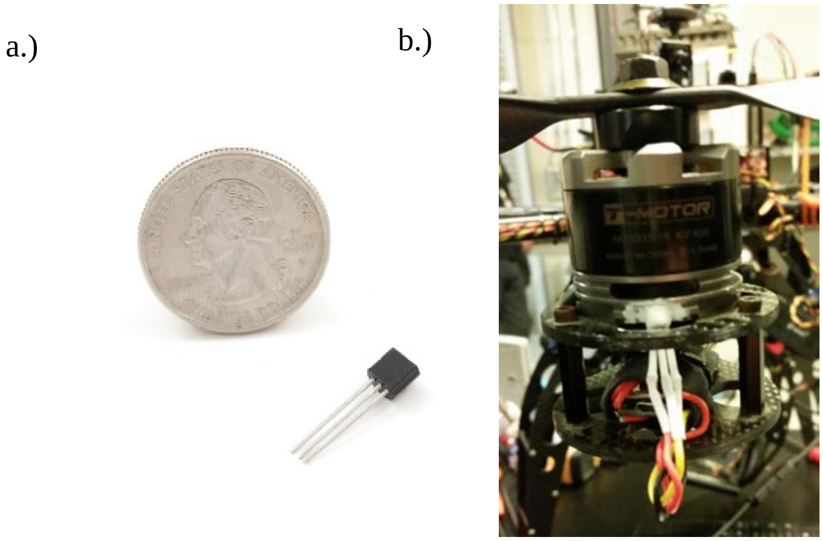


Figure 32 – a.) One Wire Digital Temperature Sensor DS18B20 [50]
b.) Temperature sensor installed on UAV motor with thermal epoxy

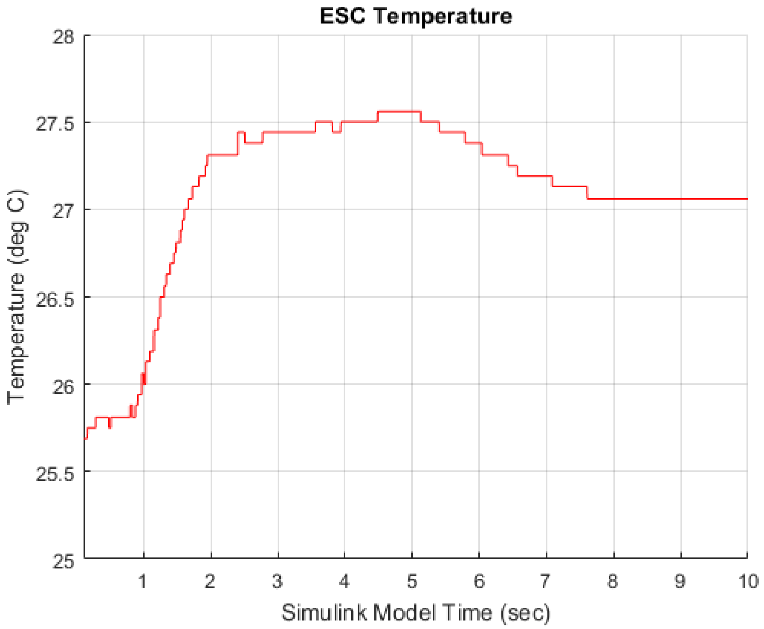


Figure 33 – Example plot of DS18B20 temperature data [41]

Next, the 90 A AttoPilot Voltage and Current Sense Breakout Board [51], shown in Figure 34b is added by splicing the power wires between the lithium polymer battery [52], shown in Figure 34c, and the ESC shown in Figure 34a. Telemetry for ESC voltage and current are fed into the Teensy and routed to Simulink for fault protection.

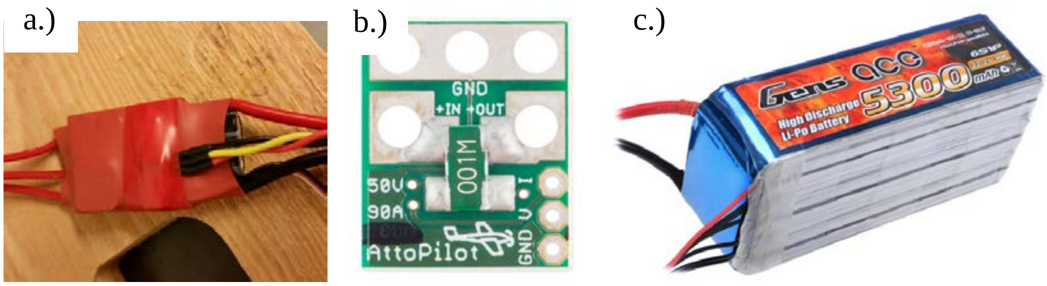


Figure 34 – a.) Electronic Speed Control (ESC) [41] b.) 90 A AttoPilot Voltage and Current Sense Breakout Board [51], c.) lithium polymer battery [52]

Once all of these sensors were added successfully, completing the sensor suite for a single arm of the copter, a second identical sensor suite was also installed on an adjacent arm of the copter as shown in Figure 35. The Teensy on one arm is designated as a central

Teensy and is connected to the Simulink FP system on the Meego. A complete connection diagram is shown in Appendix B. The Inter-Integrated Circuit (I2C) protocol is used for streaming data from each arm to the central Teensy. Although only two arms are initially instrumented, the FalconViz team can duplicate the sensor suite for each arm and deploy the system on their copters for future flights.

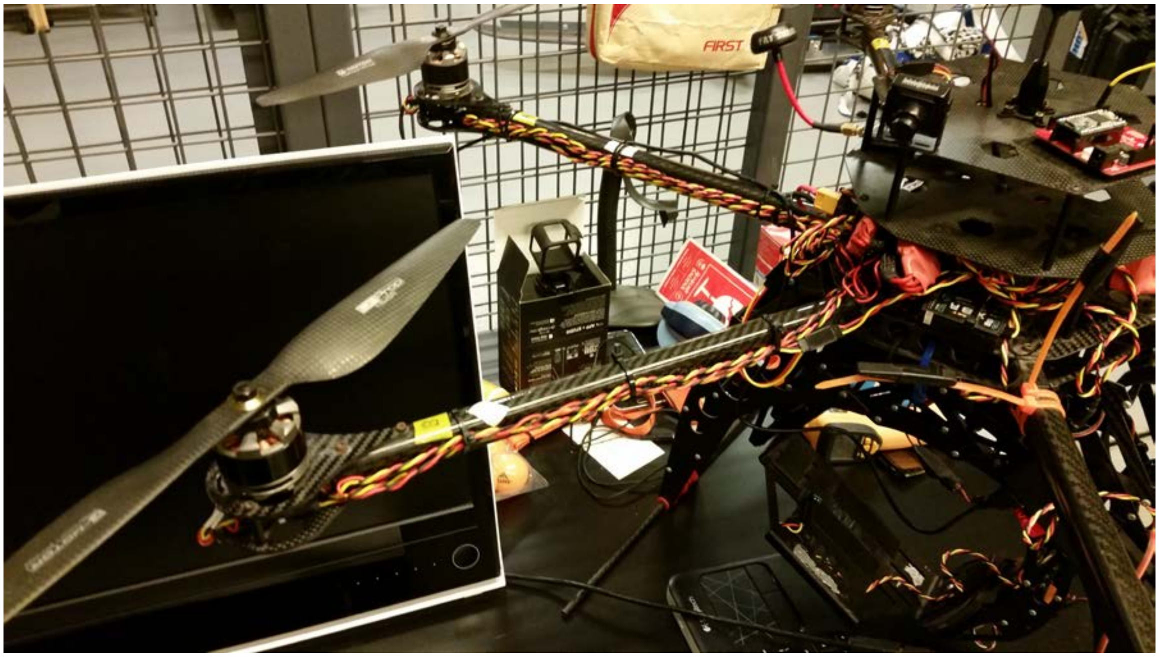


Figure 35 – Duplicate sensor suites installed on two adjacent UAV arms

The Simulink model for two arms, shown in Figure 36, is constructed by using a state machine for each sensor reading on each arm, resulting in a bank of eight state machines total, two each for acceleration, temperature, current, and voltage. All of these sensor readings are fed to the central Teensy via I2C and then to the Meego via a USB Serial connection. Each ASCII value for the sensor readings is decoded into numeric values in MATLAB. Then, the accelerometer values are fed into the KNN algorithm and the temperature, current, and voltage values are compared against minimum and maximum values, based on the limits defined in Table 3.

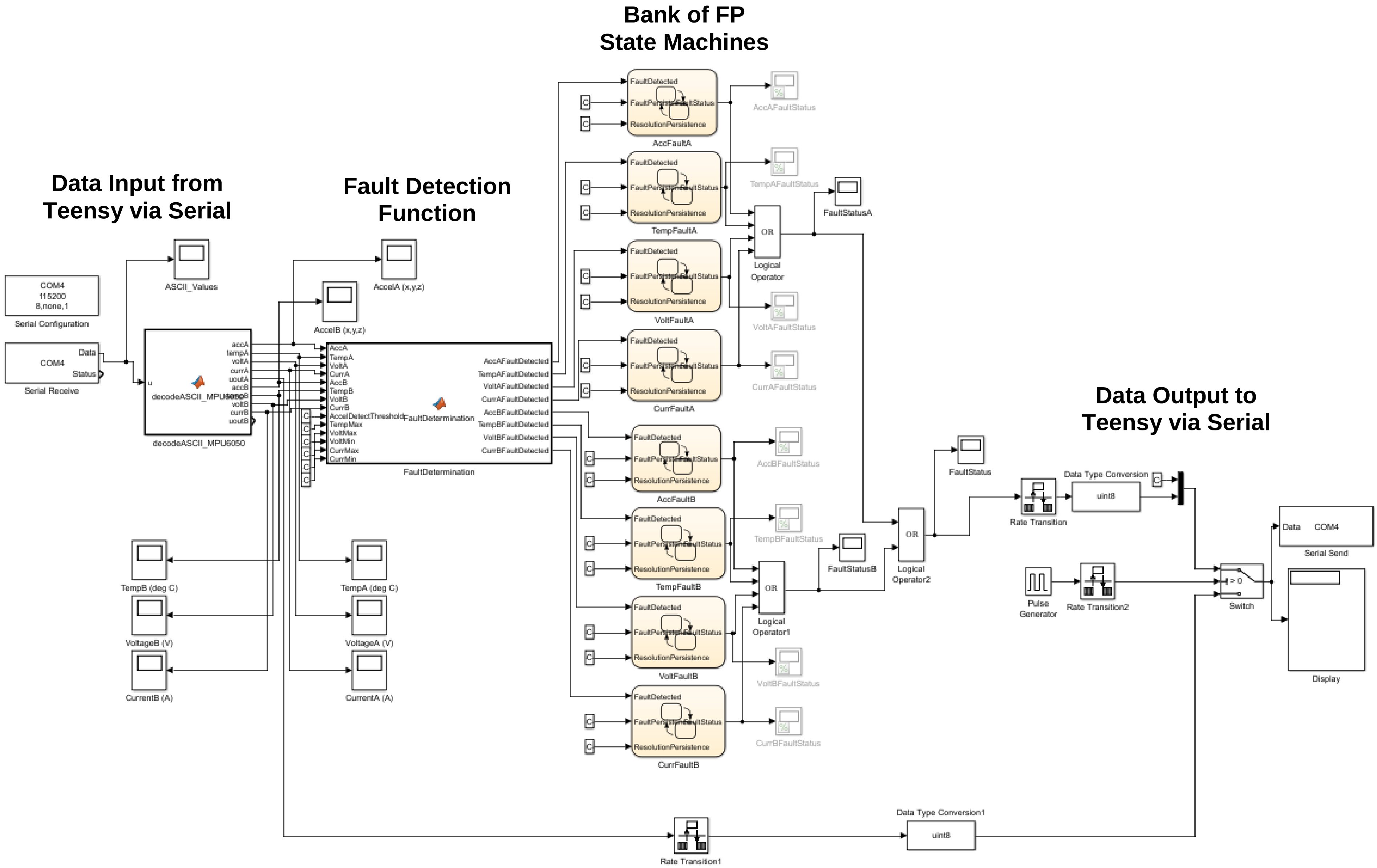


Figure 36 – Simulink model for UAV Nervous System on two arms with four sensor readings each

Table 3 – Input parameters for UAV Nervous System

Sensor Reading	**Minimum Threshold**	**Maximum Threshold**	**Time to Detect**	**Time to Resolve**
Acceleration (m/s^2)	n/a	n/a	0.025 sec	0.03 sec
Temperature (deg C)	n/a	60 deg C	1.5 sec	1.5 sec
Voltage (V)	21.6 V (6 cell*3.6 V)	25.2 V (6 cell*4.2 V)	1.5 sec	1.5 sec
Current (A)	1 A	25 A	2 sec	1.5 sec

Although the state machine in Figure 29 is used in the UAV Nervous System, it should be noted that for temperature, voltage, and current sensing a simple rule-based paradigm (with persistence) is implemented through state machines for fault detection. The acceleration detection method is a bit more complex because machine learning is used to determine if a fault has occurred or not, but ultimately the fault detection threshold is based on persistence for the outputs of the machine learning algorithm.

A series of additional test flights was completed to ensure that the upgraded UAV Nervous System was working properly. These test flights were completed with one arm first and then with both arms. Each of the thresholds for temperature, current, and voltage were tested by adjusting them slightly above and below the nominal values for each sensor. The acceleration detection was also tested, repeating the test shown in Figure 31. Because the helicopter had been in a crash since the previous flight test, it was necessary to rebalance the propellers as shown in Figure 37.

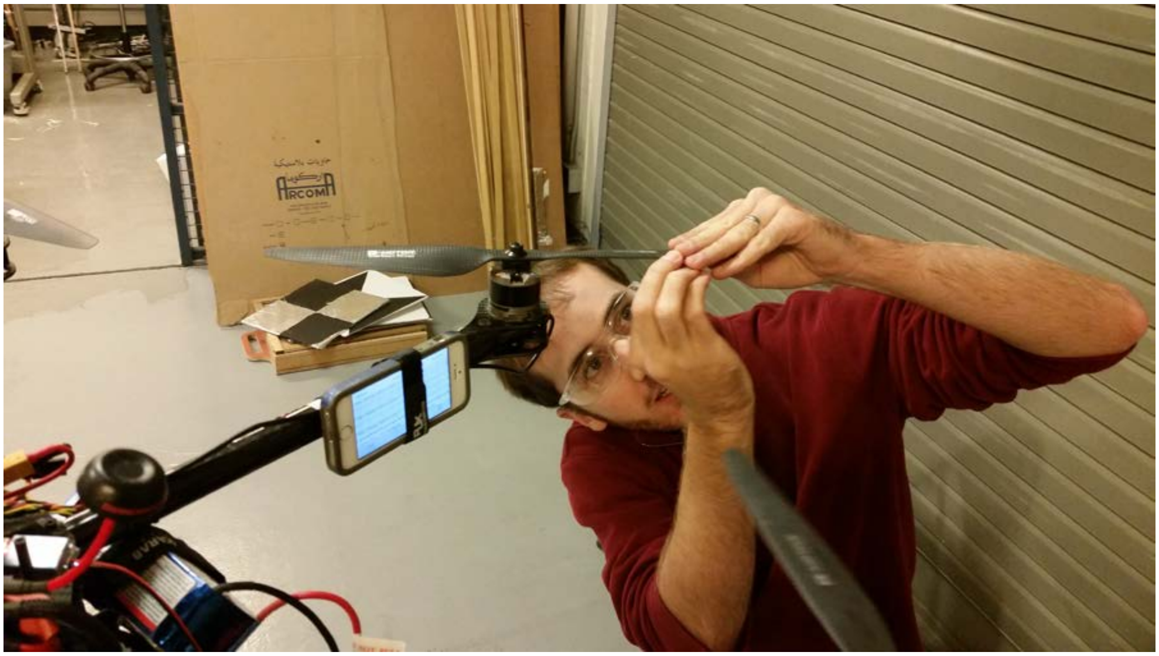


Figure 37 – Rebalancing UAV propellers using an iPhone accelerometer

3.7 Adaptation of Generic FDIR Architecture to UAV Nervous System

After the generic FDIR architecture described in Chapter 2 was created, the UAV Nervous System case study was revisited and adjusted to match the generic architecture. First, the generic functional state machine was adapted for the FalconViz UAV test flight, as shown in Figure 38. Many of the states from the generic diagram were unnecessary because of the relative simplicity of the UAV test flight. The Standby state in the bottom left is the starting state and represents the copter sitting stationary on the lab bench at the beginning of the test flight when the UAV Nervous System is activated and data recording begins. When the copter is being carried outside (`CarryingCopter=1`), the Transfer state begins. The Transfer state ends when the copter is set down on the ground outside (`ArriveAtStandoff=1`), which begins the Piloted phase. During Standoff, the first substate of the Piloted phase, the copter is sitting on the ground, waiting for the pilot's command to proceed. When the pilot begins throttling up the motors to launch the copter

(`Liftoff=1`), the NominalZone substate begins, indicating that the copter is flying. When the copter lands and the motors are powered down (`Landed=1`), the active state returns to Standoff. Note that no additional abort states are included in this functional state machine because the standard abort procedure when a fault is detected is for the pilot to land the copter.

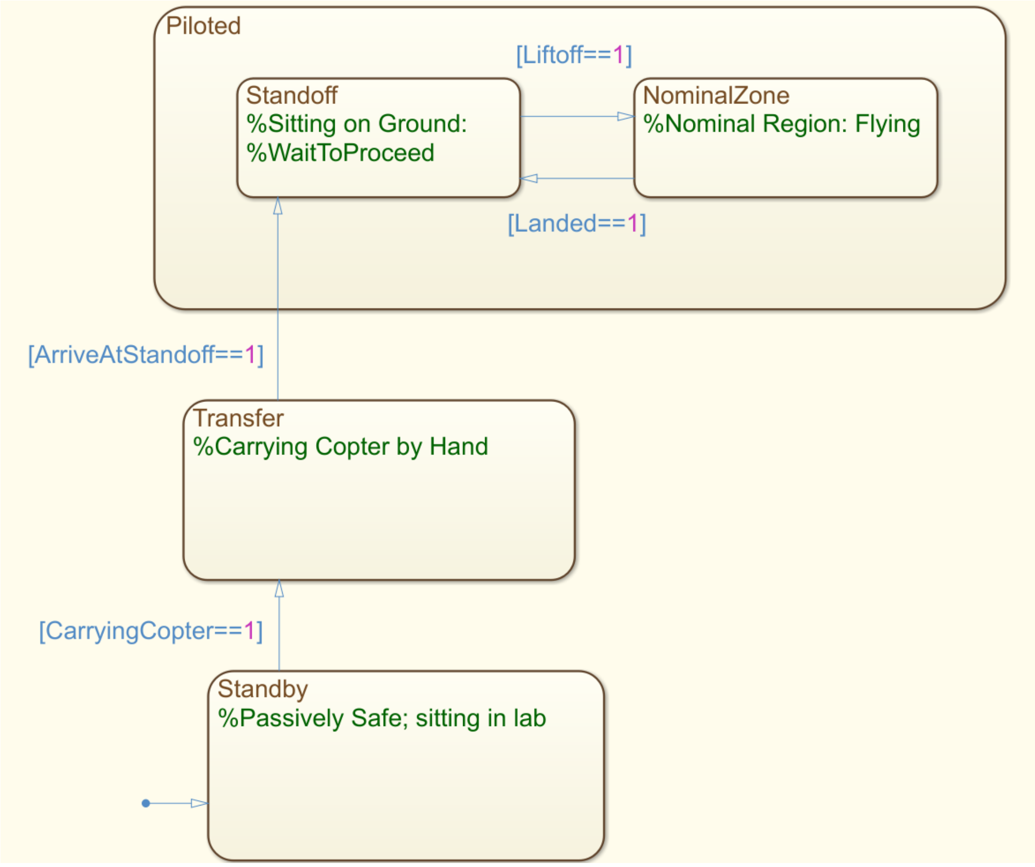


Figure 38 – Functional state machine for FalconViz UAV test flight

Next, the generic diagnostic state machine was adapted for the UAV Nervous System as shown in Figure 39. In this case, the trigger for fault detection (`FaultDetected=1`) is set to the output of the machine learning algorithm for vibration detection. This is done without regard to persistence, so whenever the machine learning algorithm indicates a fault detection, the FaultDetected state and Diagnose substate become active. Because only one fault could be caused by this particular detection, only one fault diagnosis substate is present in the diagnostic state machine. The "Propeller Unbalanced" fault is diagnosed if the fault detection trigger remains active for the "Time To Detect" of 0.025 sec from Table 3, *and* if the current state of the functional state machine is NominalZone (indicating the copter is flying). This check is evaluated by the fault diagnosis routine. The routine then sets the `FaultConfirmed1` variable to 1, causing the diagnostic state machine to enter the RespondA1 substate. The `CorrectiveAction1` variable is then set to 1, causing a signal to be sent to the pilot to land the copter as described in previous sections. If the fault detection is only intermittent, then fault diagnosis will be inconclusive. In either case, when the fault detection flag from the machine learning algorithm is set to zero for the "Time to Resolve" length of 0.03 sec the state machine will return its active state to NoFaultDetected.

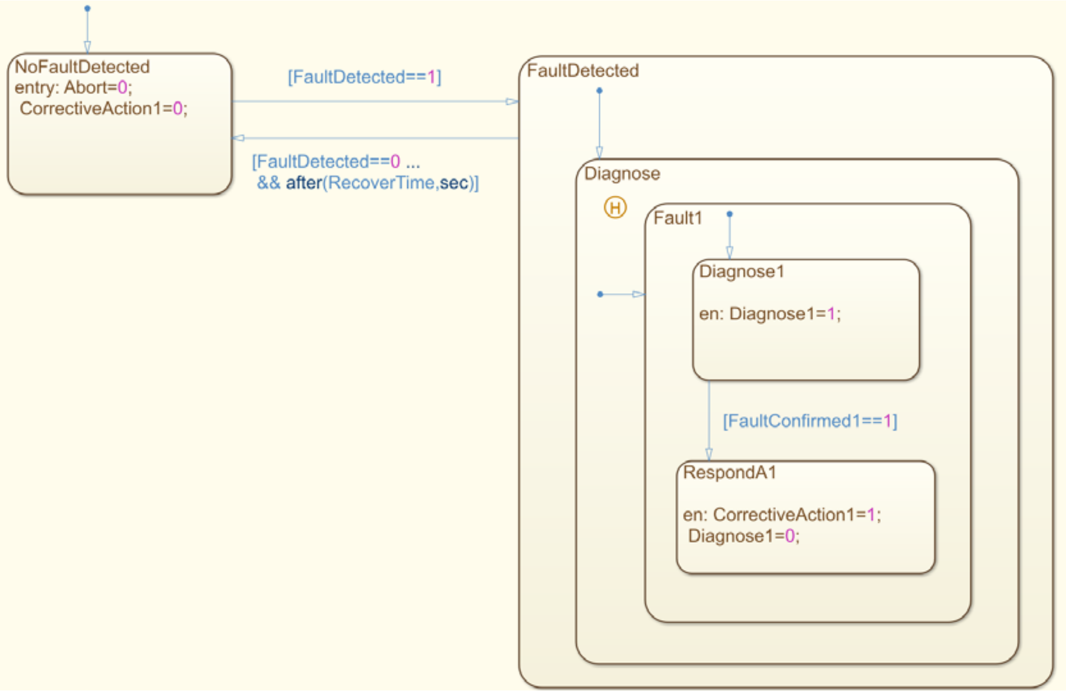


Figure 39 – Diagnostic state machine for UAV Nervous System

To demonstrate the modified FDIR architecture, recorded flight test data was loaded from a data file and replayed in Simulink using the model shown in Figure 40. The fault detection algorithm was kept unmodified from Figure 36 for one arm of the copter. The functional and diagnostic state machines described above were added, and MATLAB functions were written to calculate mode management inputs to the functional state machine and fault diagnostic inputs to the diagnostic state machine. The bank of FP state machines were kept but were used only to compare their output to the output of the new architecture. An adapted flow chart showing the interactions of the various components of the FDIR architecture for the UAV Nervous System is shown in Figure 41.

The recorded flight test data and FDIR architecture output are shown in Figure 42. Note that this flight test data is similar but not exactly the same as the flight test data shown in Figure 31; the flights were performed on different dates and at different stages of development of the UAV Nervous System. The top three plots in Figure 41 show accelerometer data and the bottom plot shows the `FaultConfirmed1` signal output by the diagnostic function in the FDIR Architecture. The data begins with the copter on the lab bench in segment A and tape is placed on the propeller to unbalance it. The copter is carried outside from the lab during segment B. The copter takes off and flies with an unbalanced propeller in segment C. The FDIR architecture quickly detects the imbalance and outputs a `FaultConfirmed` status of 1 at around 3 sec, shortly after segment C begins. At the end of segment C the copter lands, and the FDIR architecture immediately resets the fault confirmed status to 0. During segment D the copter is on the ground, and the tape is removed to restore the propeller balance. Segment E shows balanced flight, and the copter lands again at the end of segment E. During segment F, tape is added again while

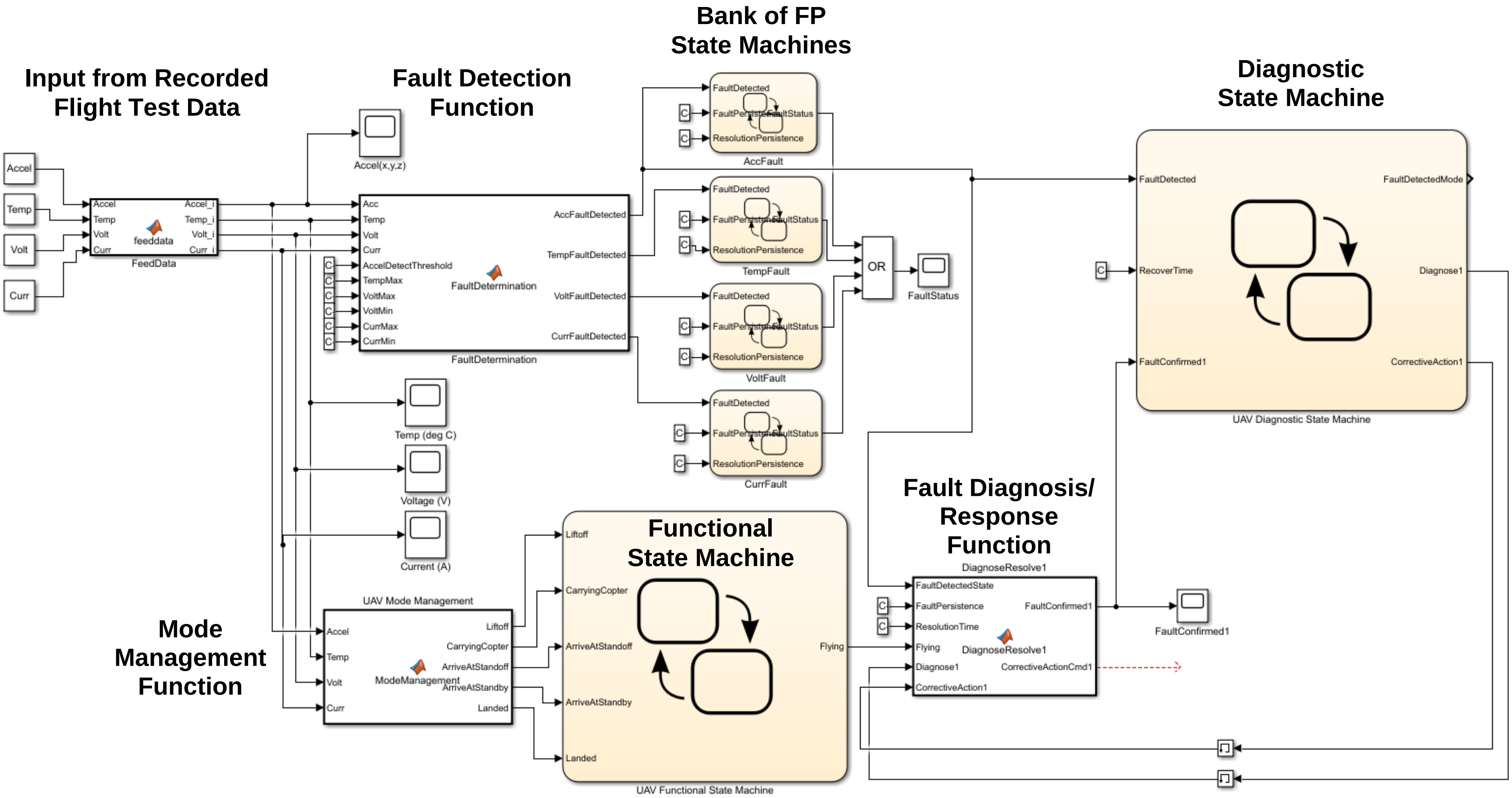


Figure 40 – Simulink diagram for modified UAV Nervous System FDIR Architecture

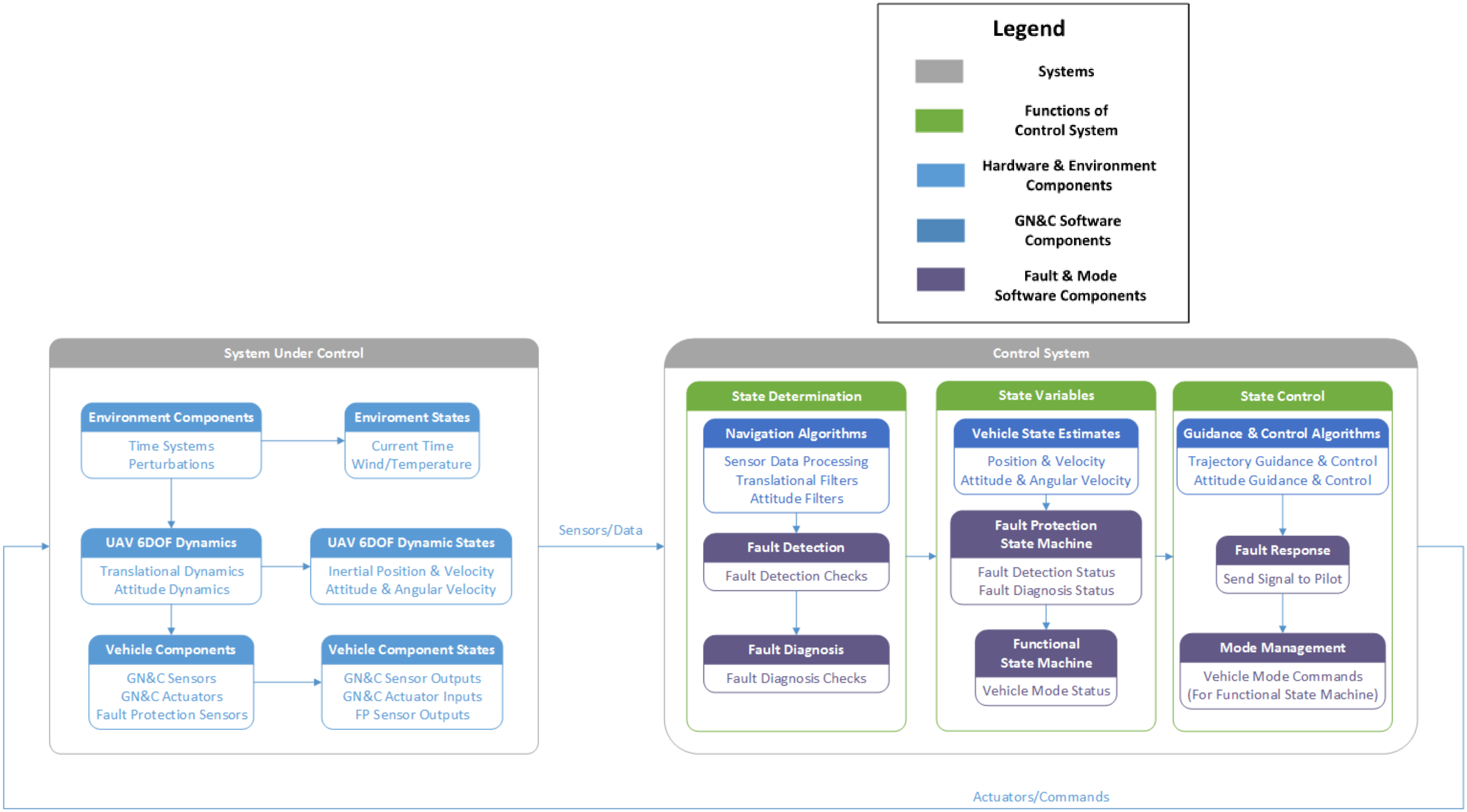


Figure 41 – Flow chart for UAV Nervous System FDIR Architecture

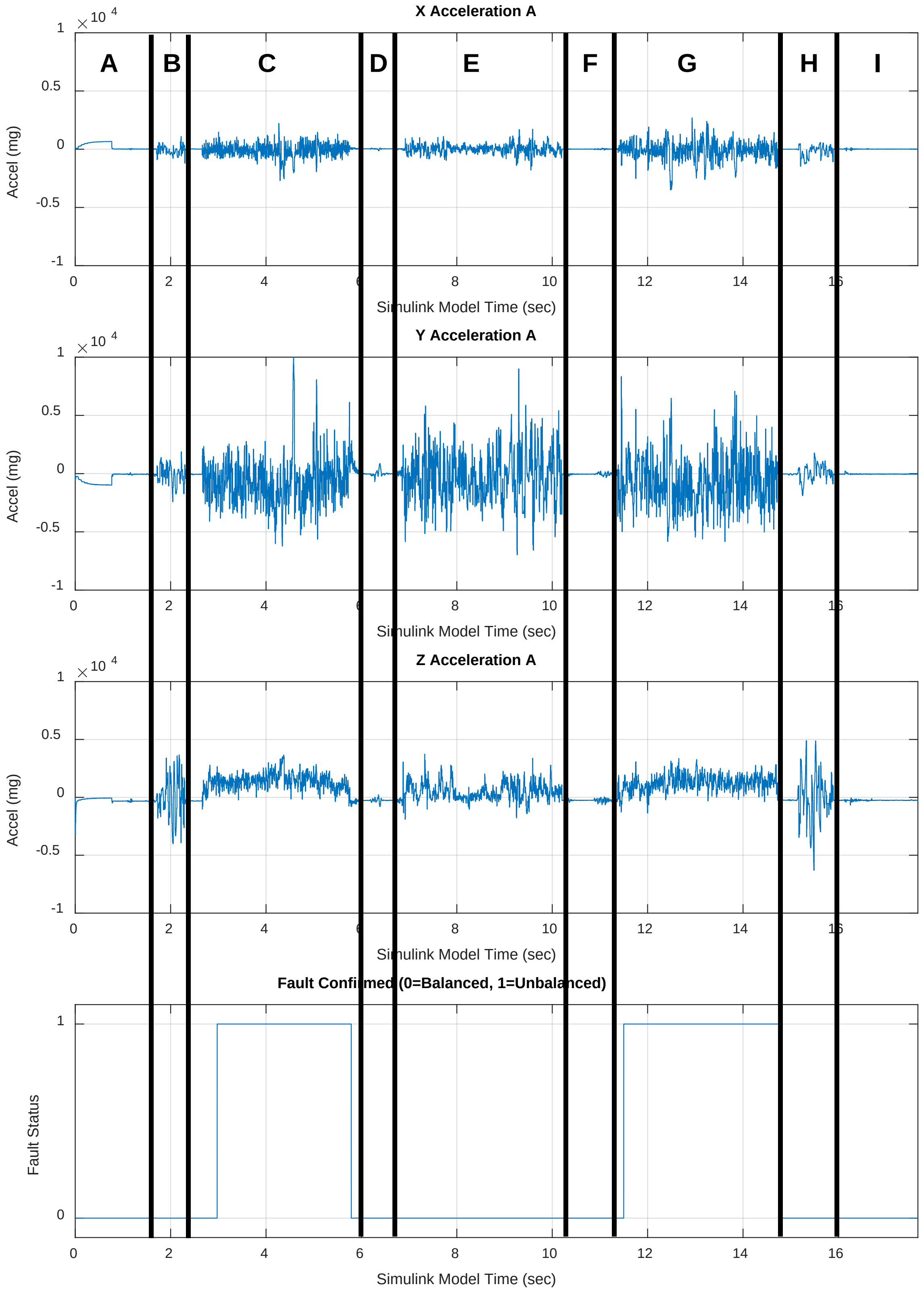


Figure 42 – Results of test flight replay with UAV Nervous System FDIR Architecture

the copter is on the ground. Unbalanced flight resumes during segment G, and between 11 and 12 sec the FDIR architecture quickly detects the imbalance and outputs a `FaultConfirmed` status of 1. The copter lands at the end of segment G, and the FDIR architecture immediately resets the `FaultConfirmed` status to 0. Note that inaccuracies in the fault status of Figure 42 are much less than in the flight test shown in Figure 31 using the previous iteration of the UAV Nervous System. Although the new system is operating on similar flight data to the previous system, the improved performance is attributable to the addition of a state machine monitoring the state of the UAV. By monitoring the ESC current, the updated architecture is able to determine whether the UAV is flying or not and takes this into account when diagnosing whether a fault is present. For example, in segment D of Figure 31 the previous system took several seconds to correctly report balanced flight, but segments D and E of Figure 42 do not exhibit the same issue.

The UAV Nervous System FDIR Architecture has been developed and tested for a terrestrial rotary wing UAV. The Nervous System utilizes a suite of sensors for two arms, each containing an accelerometer, a temperature sensor, and a voltage/current sensor. The system proof-of-concept has been shown by V&V through flight testing. The generic FDIR architecture has been successfully adapted for use with the UAV Nervous System and has been demonstrated in MATLAB/Simulink using recorded flight test data. Additional suggestions for future studies are described in Chapter 5.

CHAPTER 4. SPACE APPLICATION: MARS SAMPLE RETURN RENDEZVOUS AND ORBITING SAMPLE CAPTURE

A second use of the state machine FDIR architecture has been developed for an automated relative ProxOps application. This work supports development of a Mars Sample Return (MSR) mission. The process for development of requirements of the MSR fault protection behavior was published at the 68th International Astronautical Congress in Adelaide, Australia [53] and the Symposium on Space Innovations in Atlanta, GA [54].

4.1 Overview of Relative Proximity Operations

Relative ProxOps have been performed in Earth orbit and cis-lunar space since the early years of the space age. There is an increased need for autonomous relative ProxOps in such applications as rendezvous and docking for human space exploration and sample return missions, satellite servicing and on-orbit inspection, construction, and debris mitigation. Deep space proximity operations applications require advanced autonomy and fault protection due to the significant round-trip light time from Earth. Many autonomous ProxOps systems have been developed and proven in flight for large and complex systems [55-57] and are increasingly being used with small satellites [58,59], including the Prox-1 mission being developed at Georgia Tech [15,31]. One researcher has conducted a detailed survey of the history of relative ProxOps and describes many other recent missions [60]. Development of spacecraft GN&C software architectures often involves bringing together individually developed algorithms and evaluating them using simulation and Software-in-the-Loop (SITL) or Processor-in-the-Loop (PITL) testing [61]. The process of spacecraft GN&C algorithm development and integration using model-based design in Simulink and later autocoding into FSW has been exercised by the Orion spacecraft team at NASA's

Johnson Space Center [18]. Other missions have used selective autocoding of GN&C algorithms into FSW [62-64].

4.2 Overview of Mars Sample Return Mission Concept

The top priority stated in the current planetary science decadal survey is to perform a MSR mission [65]. NASA, JPL, and the European Space Agency are in the process of formulating a series of missions that will culminate in the return of scientifically selected Mars samples to Earth [66]. In one mission concept, a Mars orbiter would rendezvous with a sample canister launched from the surface and capture it for return to Earth [67]. The last stage of this rendezvous operation, including capture, must be autonomous. During this autonomous phase, fault protection should be used to ensure mission success [68].

In the current MSR mission concept, the Mars 2020 rover will collect soil and rock samples and cache them on the Martian surface. They will then be collected by a subsequent “fetch” rover (or other vehicle) and inserted into an Orbiting Sample container (OS). The OS will be placed on a Mars Ascent Vehicle (MAV) and launched into Mars orbit, as shown in Figure 43a. Once the MAV reaches orbit, it will release the OS into a passive near-circular orbit. The Sample Return Orbiter (SRO) will perform ground-in-the-loop rendezvous with the OS, as shown in Figure 43b, followed by autonomous approach and capture operations (also known as “terminal rendezvous and capture”) to collect the OS, as shown in Figure 43c [67]. The final approach will encompass approximately the last 100 meters of rendezvous and must be autonomous due to communication time delays for signals to travel over the large distance between Earth and Mars (between 4 and 24 minutes) [69]. Finally, the samples would be returned to Earth or cis-lunar space for recovery and laboratory study. The GN&C process for the terminal rendezvous and capture phase is complex, with a number of risk areas that could result in failure to capture the OS. This mission-critical autonomous activity presents the need for a comprehensive FP approach to ensure that operations proceed under nominal conditions, take action to address certain fault conditions, or abort the capture phase.

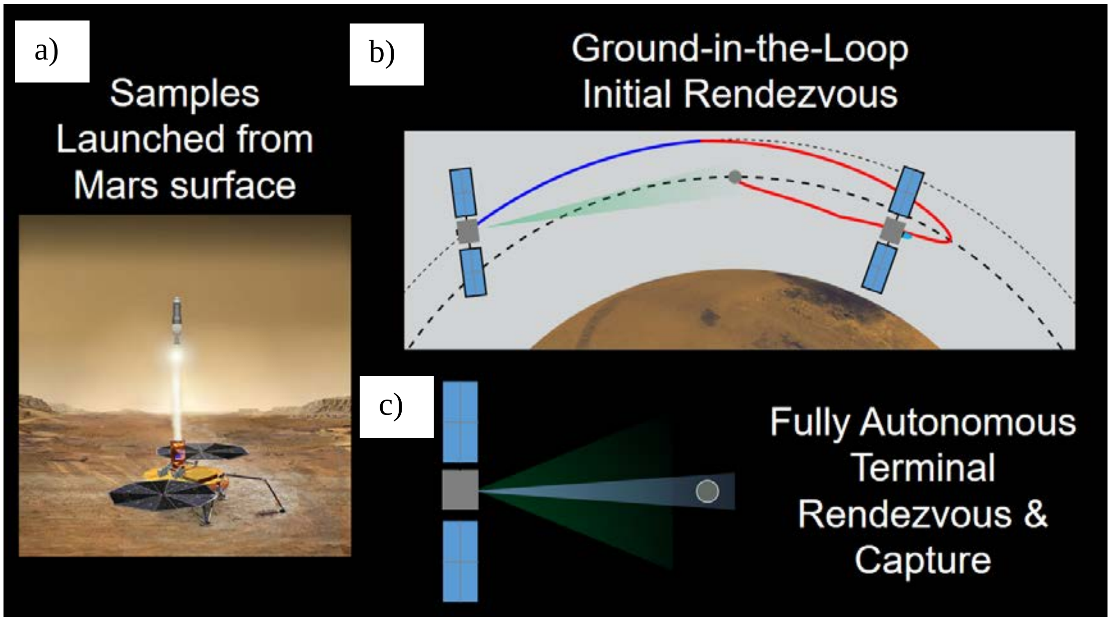

a)
Samples Launched from Mars surface
b)
Ground-in-the-Loop Initial Rendezvous
c)
Fully Autonomous Terminal Rendezvous & Capture

Figure 43 – Mars Sample Return rendezvous concept [53]

4.3 Reference Frame and Relative Orbital Element Definitions

This study makes use of several inertial and rotating reference frames, as well as dynamical parameters called relative orbital elements that are useful for visualizing the relative orbit geometry. Background information on coordinate systems and relative orbital elements is presented in this section.

Three primary reference frames are utilized in this chapter for describing orbit and attitude dynamics. The Mars International Astronomical Union (Mars IAU) reference frame is an inertial frame centered at Mars analogous to the Earth-Centered Inertial reference frame. This frame is defined by NASA's Navigation and Ancillary Information Facility at JPL for the Spacecraft, Planet, Instrument, Orientation, & Events (SPICE) ephemeris toolkit [70]:

> *Mars Mean Equator and IAU vector of J2000. The IAU-vector at Mars is the point on the mean equator of Mars where the equator ascends through the earth mean equator. This vector is the cross product of Earth mean north with Mars mean north.*

The Local Vertical Local Horizontal (LVLH) reference frame is used to describe relative orbit dynamics. The LVLH frame is an orbit-defined frame whose origin typically lies at the center of mass of the target satellite with the three axes defined by the position vector (radial), velocity vector (along-track) and their cross product (cross-track). Figure 44 shows an illustration of this coordinate frame. Note that the LVLH frame is constantly translating and rotating as the OS moves in its orbit about Mars.

For the MSR mission, it is assumed that no a priori knowledge of the OS inertial position and velocity is known. Thus, the origin of the LVLH frame is located at the OS's estimated position relative to the SRO, but the orientation of the LVLH frame is based on the SRO's inertial position. As a result, the orientation of the basis vectors $\{\hat{\boldsymbol{R}}, \hat{\boldsymbol{S}}, \widehat{\boldsymbol{W}}\}$ are defined by Eq. (1), where $\boldsymbol{r}_{SRO}$ and $\boldsymbol{v}_{SRO}$ are the positon and velocity vectors of the SRO in the Mars IAU frame and the $^{\times}$ superscript represents the skew function. When the skew

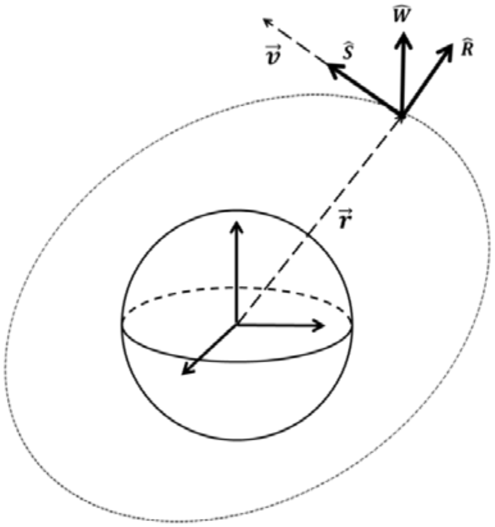


Figure 44 – Visual representation of the LVLH frame [15]

function is applied to a vector, a skew symmetric matrix is created which, when multiplied with another vector, produces the same result as a cross product between the two vectors. The following assumptions are applied in calculation of the LVLH frame unit vectors: the distance between the SRO and OS is small compared with the OS orbit radius, and the inertial orbital velocity of the OS is approximately equal to the orbital velocity of the SRO.

$$\widehat{\boldsymbol{R}} := \frac{\boldsymbol{r}_{SRO}}{||\boldsymbol{r}_{SRO}||} \qquad \widehat{\boldsymbol{W}} := \frac{\boldsymbol{r}_{SRO}^{x}\boldsymbol{v}_{SRO}}{\left\|\boldsymbol{r}_{SRO}^{x}\boldsymbol{v}_{SRO}\right\|} \qquad \widehat{\boldsymbol{S}} := \frac{\widehat{\boldsymbol{W}}^{x}\widehat{\boldsymbol{R}}}{\left\|\widehat{\boldsymbol{W}}^{x}\widehat{\boldsymbol{R}}\right\|} \tag{1}$$

The Body-Fixed Frame is a coordinate frame that is fixed with respect to the SRO. It is centered at the SRO center of mass with the three basis vectors $\{\hat{x}_b, \hat{y}_b, \hat{z}_b\}$ defined such that $\hat{y}_b$ is oriented along the imager boresight, $\hat{z}_b$ is normal to the bottom plate and oriented away from the body of the satellite, and $\hat{x}_b$ is defined by the right hand rule. The main purpose of this coordinate frame is to determine the attitude and angular velocity of the SRO relative to the Mars IAU Frame. Since the SRO does not yet have a defined shape, Figure 45 shows the orientation of the body-fixed frame with respect to the Prox-1 spacecraft as an example.

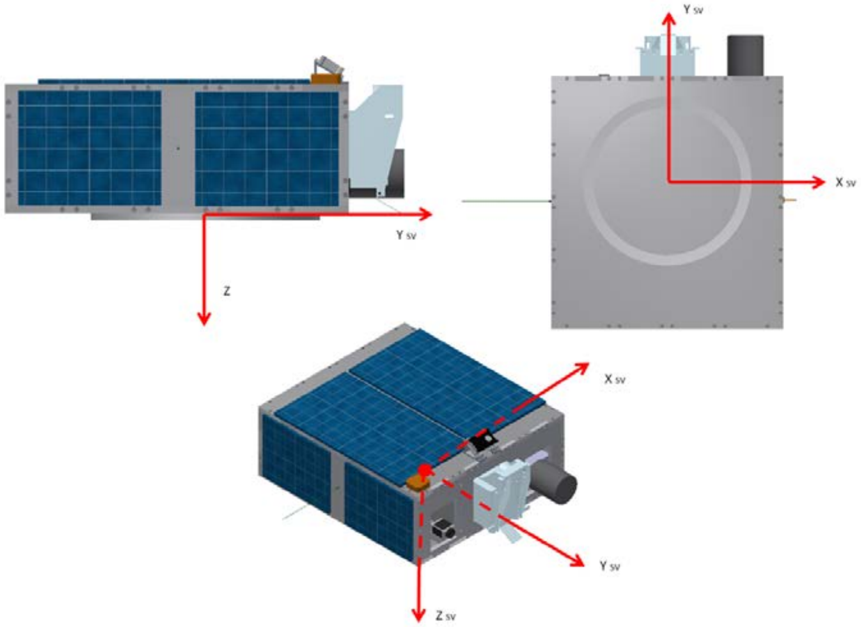


Figure 45 – Prox-1 Body-Fixed Frame orientation [15]

Relative orbital elements (ROEs) describe the relative motion of one object with respect to another in orbit, in an analogous way to the conventional orbital elements used to describe a two-body orbit [60]. ROEs were developed based on the Clohessy-Wiltshire model and rely on the corresponding assumptions of a circular target spacecraft orbit and small relative distance to the target. The ROEs described in Cartesian LVLH coordinates, where the x-axis is represented by $\boldsymbol{\hat{R}}$, the y-axis is represented by $\boldsymbol{\hat{S}}$, and the z-axis is represented by $\boldsymbol{\widehat{W}}$, are derived in detail in [71] and are given by Eqs. 2-7, where n is the mean motion of the target spacecraft, t_0 is a reference time, t is the evaluation time, x_0, y_0, z_0 are the initial LVLH position states at t_0, and $\dot{x}_0, \dot{y}_0, \dot{z}_0$ are the initial velocity states at t_0.

$$x_r = 4x_0 + \frac{2\dot{y}_0}{n} \tag{2}$$

$$y_r = y_0 - \frac{2\dot{x}_0}{n} - (6nx_0 + 3\dot{y}_0)(t - t_0) \tag{3}$$

$$a_r = \sqrt{\left(6x_0 + \frac{4\dot{y}_0}{n}\right)^2 + \left(\frac{2\dot{x}_0}{n}\right)^2} \tag{4}$$

$$E_r = atan2\left(\frac{2\dot{x}_0}{n}, 6x_0 + \frac{4\dot{y}_0}{n}\right) + n(t - t_0) \tag{5}$$

$$A_z = \sqrt{z_0^2 + \left(\frac{\dot{z}_0}{n}\right)^2} \tag{6}$$

$$\psi = atan2\left(z_0, \frac{\dot{z}_0}{n}\right) + n(t - t_0) \tag{7}$$

The orbit of the chaser spacecraft about the target is a two-by-one ellipse about the instantaneous center of motion (x_r, y_r) with a semi-major axis of a_r, a semi-minor axis of $\frac{1}{2}a_r$, and a cross-track amplitude of A_z. E_r is the relative eccentric anomaly of the ellipse, and ψ is the cross-track phase angle. Figure 46 shows this relative orbit geometry, and Eqs. 8-13 describe the LVLH Cartesian position and velocity state in terms of ROEs.

$$x = x_r - \frac{1}{2}a_r \cos(E_r) \tag{8}$$

$$y = y_r + a_r \sin(E_r) \tag{9}$$

$$z = A_z \sin(\psi) \tag{10}$$

$$\dot{x} = \frac{n}{2}a_r \sin(E_r) \tag{11}$$

$$\dot{y} = -\frac{3}{2}nx_r + na_r \cos(E_r) \tag{12}$$

$$\dot{z} = nA_z \cos(\psi) \tag{13}$$

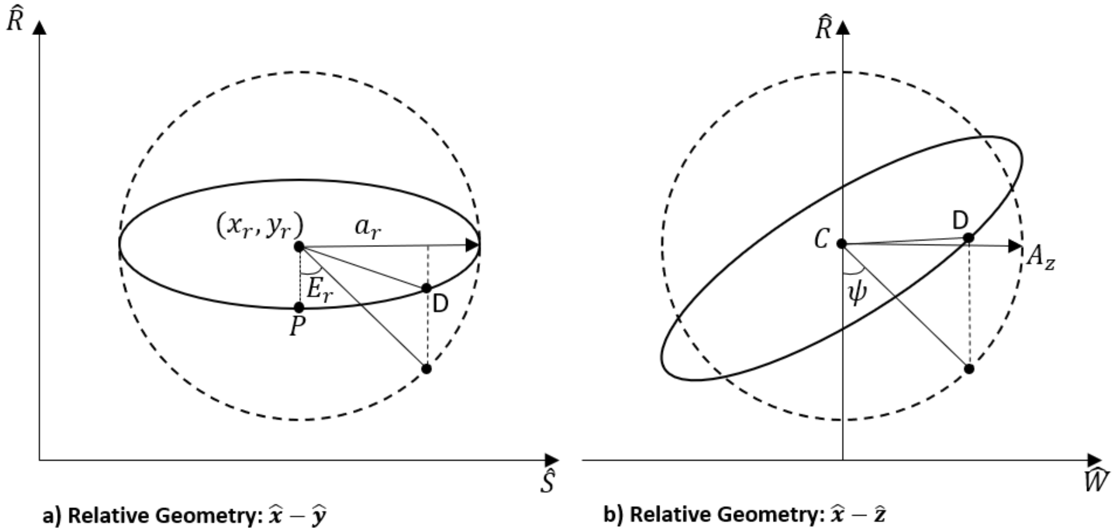
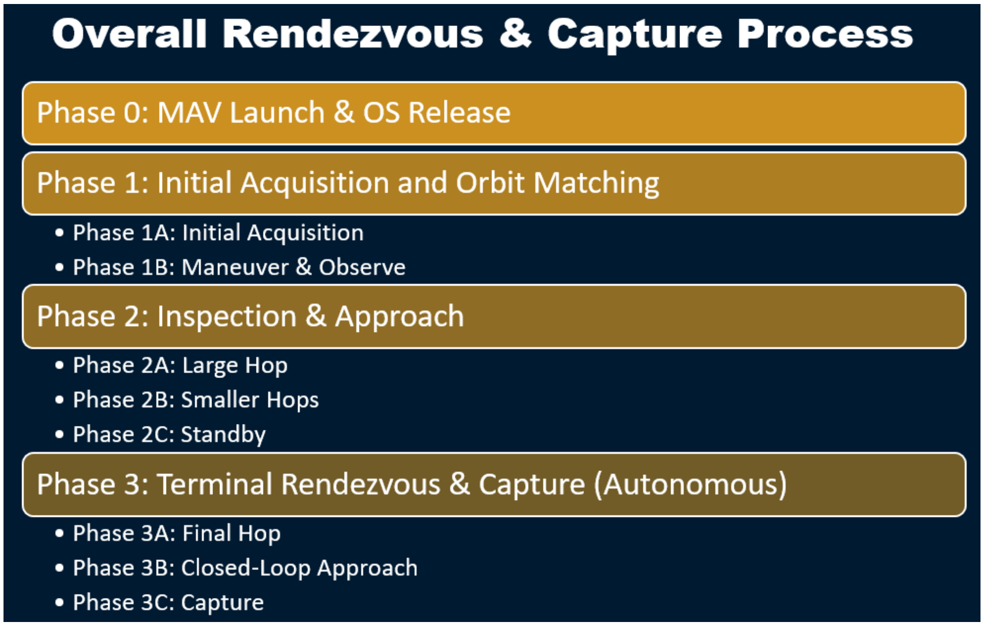

$\hat{R}$
(x_r, y_r)
a_r
E_r
D
P
$\hat{S}$
a) Relative Geometry: $\hat{x} - \hat{y}$
$\hat{R}$
C
D
A_z
ψ
$\hat{W}$
b) Relative Geometry: $\hat{x} - \hat{z}$

Figure 46 – Relative orbit geometry

4.4 Rendezvous & Capture Concept of Operations

In collaboration with the MSR rendezvous team at JPL, a detailed process has been defined for the overall MSR rendezvous and capture concept of operations. This section presents the detailed process, with particular attention to autonomous terminal rendezvous. A description of rendezvous maneuvers using ROEs and example trajectories are also presented. An overview of the full process is shown in Figure 47.



Overall Rendezvous & Capture Process

Phase 0: MAV Launch & OS Release

Phase 1: Initial Acquisition and Orbit Matching

- Phase 1A: Initial Acquisition
- Phase 1B: Maneuver & Observe

Phase 2: Inspection & Approach

- Phase 2A: Large Hop
- Phase 2B: Smaller Hops
- Phase 2C: Standby

Phase 3: Terminal Rendezvous & Capture (Autonomous)

- Phase 3A: Final Hop
- Phase 3B: Closed-Loop Approach
- Phase 3C: Capture

Figure 47 – Overall rendezvous and capture process

4.4.1 Initial Rendezvous Process

Phase 0 of the rendezvous process involves launching the MAV from the surface of Mars and releasing the OS into orbit around Mars. Phase 1 is initial acquisition and orbit matching, in which the SRO first acquires images of the OS from thousands of kilometers away. Relative orbit determination is performed by engineers on the ground, and once a reliable relative orbit estimate is obtained maneuvers are performed to begin matching the SRO orbit to within about 10 km of the OS. During Phase 2, the OS is continually inspected by the SRO to obtain a refined relative orbit estimate with ground-in-the-loop navigation solutions. Approach maneuvers or "hops" are also performed during Phase 2 to gradually move the SRO closer to the OS. This study begins in Phase 2C, after the SRO is placed in a passively safe "standby" orbit within 100 m of the OS. This orbit is an inclined "safety ellipse" in either a leading or trailing orbit relative to the OS. As shown in Figure 48, the relative orbit has an out-of-plane cross-track component. If there is any altitude difference between the SRO's relative center of motion and the OS, the relative orbit of the SRO may drift along-track toward or away from the OS. There is no risk of collision if drift occurs because the SRO's relative orbit is not in-plane with the OS.

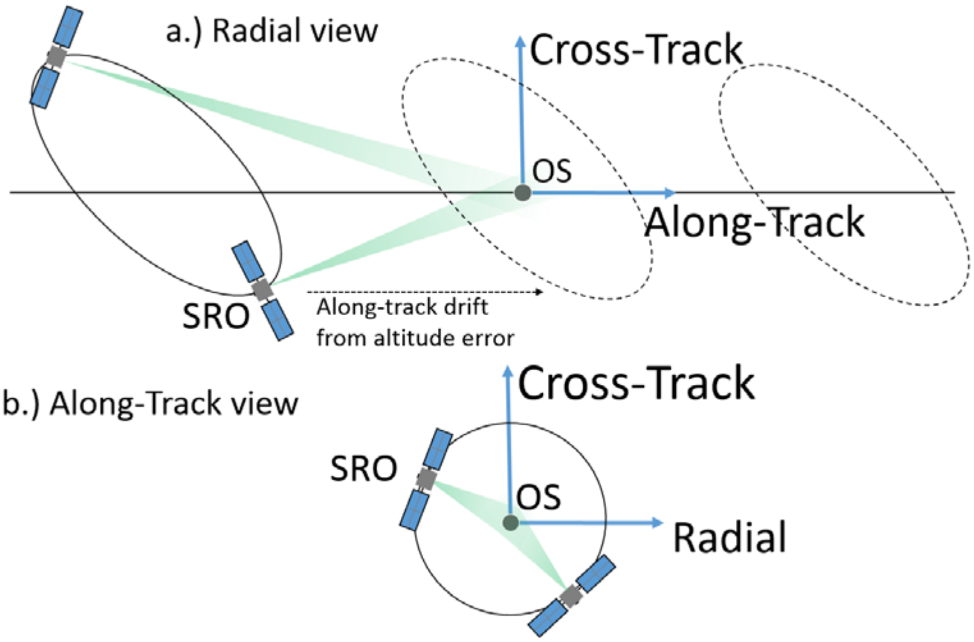


Figure 48 – Illustration of passively safe standby trajectory (not to scale)
a.) Radial view (looking towards Mars) b.) Along-track view (along OS velocity vector)

4.4.2 Autonomous Terminal Rendezvous & Capture Process

The terminal rendezvous and capture process in Phase 3 was developed into a functional state machine, shown in Figure 49. This state machine is based on the generic functional state machine and is used by the fault protection system to determine how to respond to various faults as they are detected. This section walks through each step of this process and explains how the state machine operates. Two types of approach strategies are currently under consideration for the terminal rendezvous phase. The first is a forced motion approach, which involves approaching the target using autonomous closed-loop control either via the along-track direction (v-bar) [56], radial direction (r-bar) [55], or

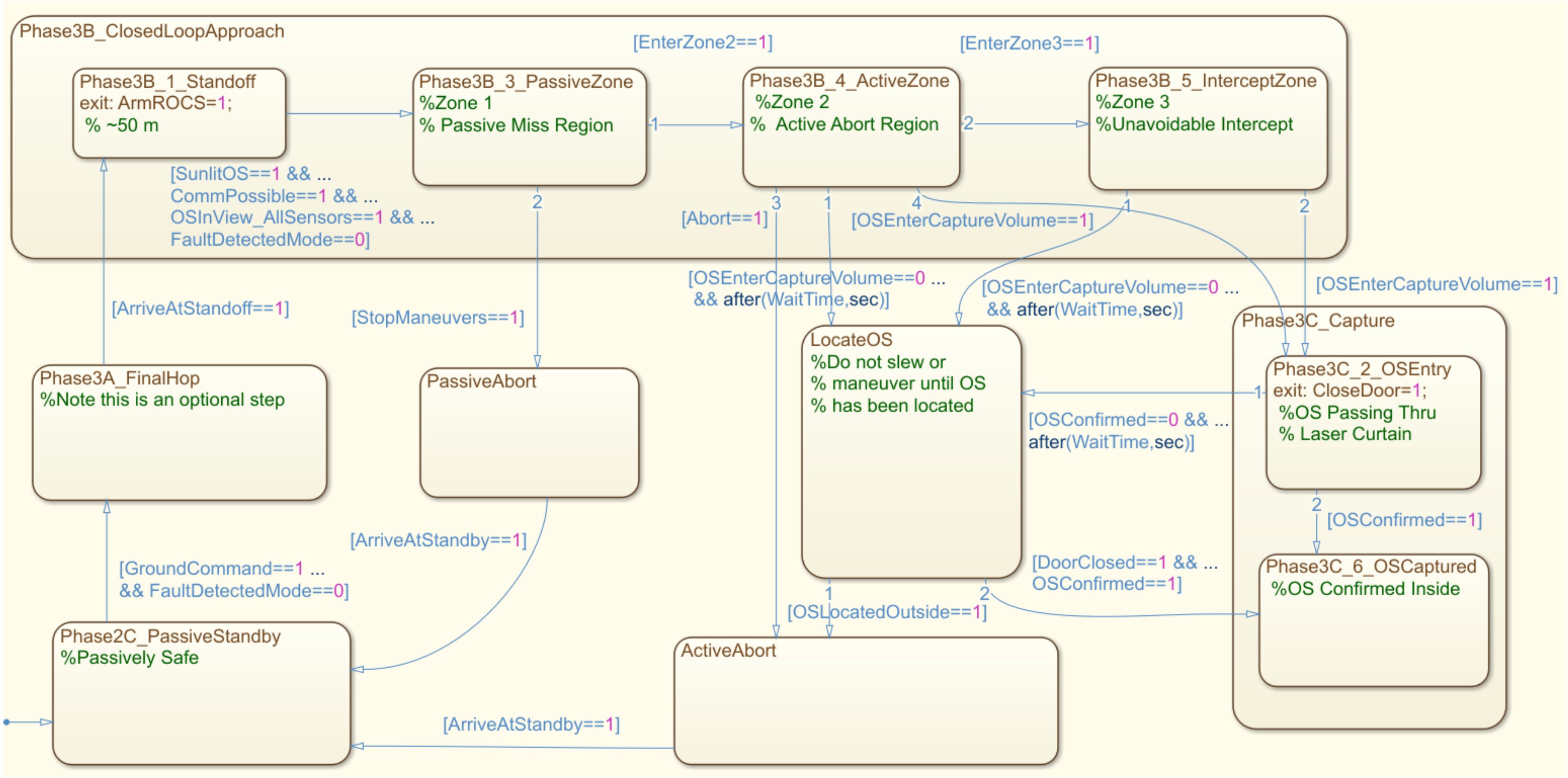


Figure 49 – State machine for rendezvous and capture process

some hybrid between the two directions. The second strategy is a ballistic approach, which involves performing a single maneuver to place the spacecraft on a ballistic collision course. The ballistic approach is computationally simpler but has fewer safety considerations. This study assumes a forced motion v-bar approach, but differences between the forced motion and ballistic approach will be mentioned in the description of the rendezvous process.

Fault responses are calibrated based on the relative risk to the mission in each sub-phase. The state machine in Figure 49 represents both nominal and off-nominal processes. The flow of the nominal process begins in the bottom left corner and continues upwards around the border of the chart to the bottom right. Off-nominal processes are shown in the middle of the chart.

4.4.2.1 Phase 2C: Passive Standby

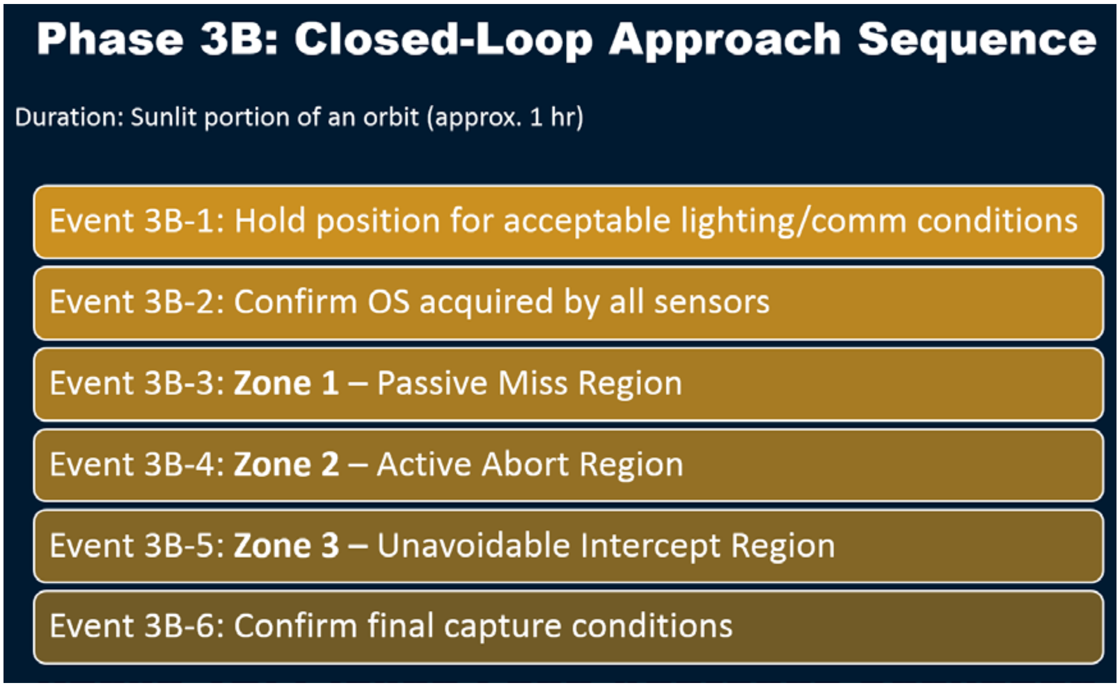
The SRO begins in a passively safe standby orbit at the end of the ground-in-the-loop rendezvous process, as described in Section 4.4.1; this is the state in the bottom left of Figure 49. The SRO can remain in Phase 2C for an extended period of time if necessary. A ground command must be provided (`GroundCommand=1`) to initiate the autonomous sequence (Phase 3). Even when a ground command is provided, the autonomous sequence will not initiate if a fault has been detected (`FaultDetectedMode=1`).

4.4.2.2 Phase 3A: Final Hop

If a forced motion approach strategy is chosen, the autonomous sequence would include a Final Planar Hop from the out-of-plane passively safe trajectory to the plane of the terminal approach corridor, removing the cross-track motion provided by the safety ellipse while also moving closer to the target in the along-track direction. The planar hop ends when the SRO arrives at a standoff position (`ArriveAtStandoff=1`) just before the start of the final approach. Note that the Final Hop phase is not necessary if a ballistic approach strategy is chosen.

4.4.2.3 Phase 3B-1: Standoff

The closed-loop approach sequence (Phase 3B) is shown in Figure 50. This phase begins with the SRO holding position in a standoff tens of meters away from the OS on the along-track axis. When proper lighting (`SunlitOS=1`) and communication conditions (Earth not occulted by Mars, `CommPossible=1`) are achieved and the OS has been acquired by all rendezvous sensors that can see it at that range (`OSInView_AllSensors=1`), the Rendezvous OS Capture System (ROCS) capture mechanism is armed (`ArmROCS=1`) and the approach begins. Even if all of these conditions are met, the final approach will not initiate if a fault has been detected (`FaultDetectedMode=1`). Although real-time two-way communication is not possible during autonomous terminal approach and capture (Phase 3) because of communication time delays, the communication condition is imposed to allow the ground team to monitor autonomous operations by streaming telemetry and possibly live video to the ground.



Phase 3B: Closed-Loop Approach Sequence

Duration: Sunlit portion of an orbit (approx. 1 hr)

Event 3B-1: Hold position for acceptable lighting/comm conditions

Event 3B-2: Confirm OS acquired by all sensors

Event 3B-3: **Zone 1 –** Passive Miss Region

Event 3B-4: **Zone 2 –** Active Abort Region

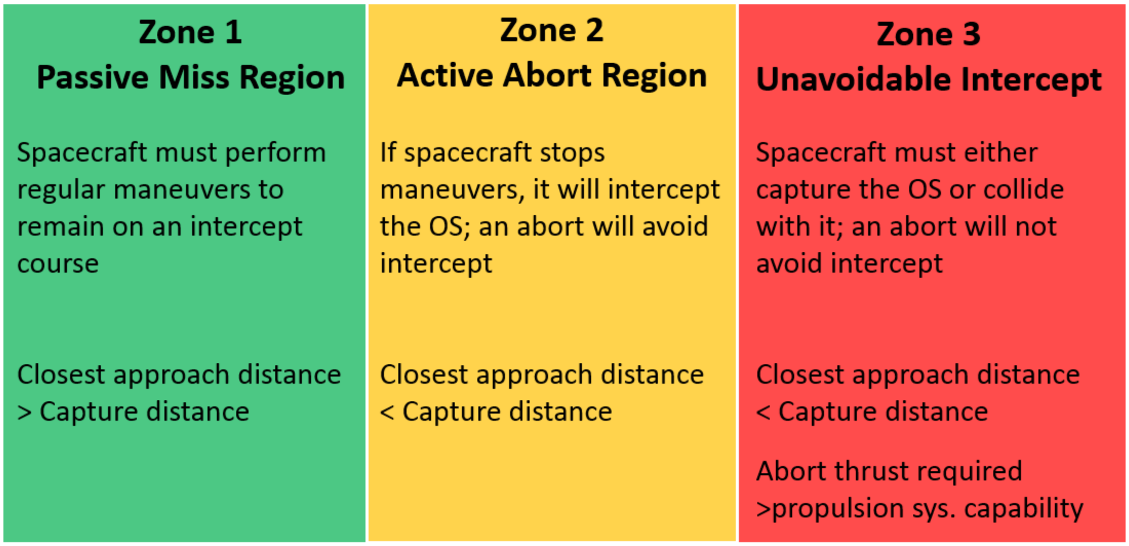
Event 3B-5: **Zone 3 –** Unavoidable Intercept Region

Event 3B-6: Confirm final capture conditions

Figure 50 – Closed-Loop Approach Sequence

4.4.2.4 Phase 3B-3: Zone 1 – Passive Miss Region

"Zones of Criticality" have been specified for the final approach after leaving the standoff position. The zones shown in Figure 51 are used to alter fault protection behavior based on distance to the target and time to intercept. Durations and distances listed are dependent on the rendezvous approach strategy and specific parameters selected, so the transition conditions between these zones may vary, but the criticality (and thus impact on FP behavior) of the zones will endure regardless of the implementation selected. Also note that Zone 1 (Passive Miss Region) does not exist if a ballistic terminal approach is selected. The closest approach distance and minimum velocity used to calculate the zone transitions are computed by propagating the ROEs after each maneuver using Clohessey-Wiltshire assumptions and then converting to Cartesian coordinates.



Zone 1 Passive Miss Region	Zone 2 Active Abort Region	Zone 3 Unavoidable Intercept
Spacecraft must perform regular maneuvers to remain on an intercept course	If spacecraft stops maneuvers, it will intercept the OS; an abort will avoid intercept	Spacecraft must either capture the OS or collide with it; an abort will not avoid intercept
Closest approach distance > Capture distance	Closest approach distance < Capture distance	Closest approach distance < Capture distance Abort thrust required >propulsion sys. capability

Figure 51 – Notional “zones of criticality”

Zone 1 is called the “passive miss region”. During this zone the SRO must perform regular maneuvers to remain on an intercept course with the OS. If the SRO stops maneuvering (a passive abort), then it will pass by the OS harmlessly.

4.4.2.5 Passive Abort

If at any point in the Passive Miss Region something goes wrong, the system simply stops maneuvers (`StopManuevers=1`) and enters Passive Abort, allowing the SRO to drift away from the OS. Once the SRO has reached a safe distance from the OS (`ArriveAtStandby=1`), it returns to the out-of-plane Passive Standby trajectory and awaits ground commands before resuming autonomous operations.

4.4.2.6 Phase 3B-4: Zone 2 – Active Abort Region

If no problems occur during the Passive Miss Region, the system will enter the “Active Abort Region” when the minimum propagated range from the SRO to the OS becomes less than the defined capture distance from the SRO’s center of mass. During this zone, if the SRO stops maneuvers it will intercept the OS, but if an active abort maneuver is performed intercept can be avoided.

4.4.2.7 Active Abort

If at any point during the Active Abort Region something goes wrong, an abort can be commanded (`Abort=1`) to return to Passive Standby via the Active Abort mode. The active abort maneuver immediately adds out-of-plane motion and moves away from the target to avoid an intercept. It then allows the SRO to drift slowly away from OS until it returns to the out-of-plane Passive Standby trajectory (`ArriveAtStandby=1`).

4.4.2.8 Phase 3B-5: Zone 3 – Unavoidable Intercept Region

Finally, just before intercept the system enters the third zone, which is called the “unavoidable intercept region”. The condition to enter Zone 3 is that the abort thrust required must be higher than the SRO’s propulsion system can generate, in addition to the minimum propagated range from the SRO to the OS being less than the defined capture distance from the SRO’s center of mass. During this zone, the SRO can no longer avoid intercepting the OS even if an abort maneuver is performed. It must either capture the OS, or it will likely collide. Note that Zone 3 may be very short (on the order of seconds) if the SRO has a robust thrust capability.

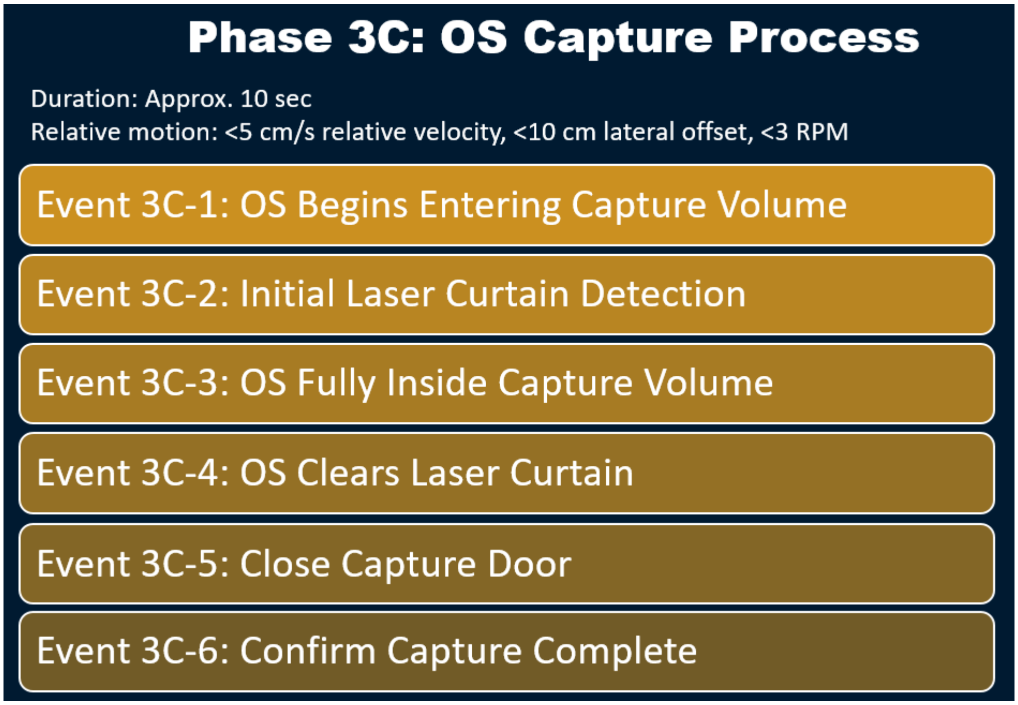
4.4.2.9 LocateOS

If capture is unsuccessful and the OS does not enter the capture volume (`OSEnterCaptureVolume=1`) within the specified WaitTime, the system enters the “LocateOS” state. It will attempt to determine where the OS is located

(`OSLocatedOutside=1`) before performing any slew or thrust maneuvers. Once the OS is found (`OSConfirmed=1`), an abort maneuver is commanded (`Abort=1`).

4.4.2.10 Phase 3C: Capture

If the OS enters the capture volume successfully, the capture process (Phase 3C) begins. This process is shown in Figure 52. The OS passes by a sensor such as a laser curtain, which indicates that it is entering the capture volume. Once the OS has cleared the laser curtain and is fully inside the capture volume, a command is sent to close to capture door (`CloseDoor=1`). A confirmation sensor then verifies that the OS is inside (`OSConfirmed=1`). If the OS cannot be confirmed inside the capture volume within the specified `WaitTime` after the door has closed, the system enters the LocateOS state and commands an abort (`Abort=1`).



Phase 3C: OS Capture Process

Duration: Approx. 10 sec
Relative motion: <5 cm/s relative velocity, <10 cm lateral offset, <3 RPM

Event 3C-1: OS Begins Entering Capture Volume

Event 3C-2: Initial Laser Curtain Detection

Event 3C-3: OS Fully Inside Capture Volume

Event 3C-4: OS Clears Laser Curtain

Event 3C-5: Close Capture Door

Event 3C-6: Confirm Capture Complete

Figure 52 – OS Capture Process

4.4.3 Description of Terminal Rendezvous Maneuvers

While there are many different approach schemes that can be used for terminal rendezvous, a v-bar approach has been selected to create scenarios to test the FP system. This approach begins with the SRO ahead of the OS in the same orbit. The semi-major axis of the SRO's orbit is then slightly reduced so that the two spacecraft will drift apart further if a maneuver cannot be performed. Reducing the semi-major axis results in an eccentricity change, creating the appearance of "hops" in the LVLH plane. The SRO then executes a maneuver in the radial direction to begin another hop every time it crosses the orbit of the OS. ROEs are used to visualize and design the approach and to provide an initial estimate of the velocity change (ΔV) required for each maneuver.

The ΔV required to cancel out the cross-track velocity is found with the following procedure. First, z is set to zero in Eq. 10, the cross track phase angle ψ is obtained, and this value and the current Cartesian LVLH state are substituted into Eq. 7 to obtain the time of xy-plane crossing. The ΔV applied at this time is the negative (-) of Eq. 13. This maneuver marks the beginning of the Final Hop. At the end of the Final Hop, a "hold position" maneuver is performed to hold the relative position constant. In this maneuver, all relative velocity is cancelled out when the chaser crosses the LVLH x-axis. At any time, the relative drift of the spacecraft in the along-track direction can be stopped by setting x_r to zero (giving both spacecraft the same semi-major axis) and solving Eq. 2 for $\dot{y}_0$. This is known as a "freeze drift" maneuver.

When defining the v-bar approach maneuvers, ROEs are helpful in describing the size of each hop. The ROE approach parameters $x_{r,app}$ and $a_{r,app}$ completely describe the distance that will be traveled in each hop, the time each hop will take, and the rate at which the along-track center of motion of the SRO will move away from the OS in the event of a passive abort. The time required for each hop, Δt, is given in Eq. 14. This equation is obtained by setting Eq. 8 to zero and substituting the chosen values of $x_{r,app}$ and $a_{r,app}$. The along-track distance traveled in each hop, Δy, is given in Eq. 16. This equation is obtained by substituting Eq. 14 into Eq. 15 then substituting the result into Eq. 9. Eq. 15 is a redefinition of Eq. 3 in terms of ROEs rather than Cartesian coordinates.

$$\Delta t = \frac{2}{n}\left(\pi - \text{acos}\left(\frac{2x_{r,app}}{a_{r,app}}\right)\right) \tag{14}$$

$$y_r = y_{r_0} - \frac{3}{2}nx_{r,app}(\mathrm{t} - \mathrm{t}_0) \tag{15}$$

$$\Delta y = \ -3x_{r,app}\left(\pi - \text{acos}\left(\frac{2x_{r,app}}{a_{r,app}}\right)\right) - 2a_{r,approach}\sin\left(\text{acos}\left(\frac{2x_{r,app}}{a_{r,app}}\right)\right) \tag{16}$$

4.4.4 Example Mission Scenarios

Plots of example trajectories are shown in this section to illustrate a nominal terminal rendezvous approach, a passive abort scenario, and an active abort scenario. These plots have been generated by MATLAB code assuming linear relative dynamics and impulsive ΔV maneuvers. For each scenario, an *xy*-projection of the trajectory, a three-dimensional (3D) view, and time histories of the relative position and velocity are shown. Impulsive maneuvers are shown in the *xy*-projection and the 3D view as red vectors indicating location, ΔV direction, and ΔV magnitude. Note that ΔV magnitudes are scaled automatically in each plot by MATLAB’s quiver3 function for visibility, so they only have meaning relative to one another in the same plot and their exact magnitudes cannot be read directly from an individual plot or compared directly with other plots. Red diamonds indicate the location of maneuvers in the time history plots. Impulsive maneuvers are represented as discontinuities in the velocity time history plot.

The SRO initial orbit conditions for all scenarios are expressed in terms of ROEs. The conditions are given by setting x_r to 0 m, y_r to 50 m, a_r to 20 m, A_z to 10 m, E_r to zero, and ψ to zero. For the v-bar approach, $x_{r,approach}$ is set to -0.5 m and $a_{r,approach}$ is set to 4 m. These parameters result in a Δt of 4,251 sec (70.85 min) and a Δy of -10.48 m per hop, with the negative indicating a decreasing y_r with each hop.

4.4.4.1 Nominal Approach Trajectory

A nominal approach trajectory is shown in Figures 53-56 and involves the following phases. First, out-of-plane natural motion occurs in the passively safe standby trajectory before any control is activated; this is the blue portion of the trajectory in the plots. Once trajectory control is activated (at the start of the black portion of the trajectory), the controller allows the SRO to continue in natural motion until the xy-plane is reached. At this point, a planar hop maneuver is commanded to remove all out-of-plane motion and the red portion of the trajectory begins. The controller allows the SRO to continue coasting until the along-track axis is reached. At this point, a hold position maneuver is commanded to hold the SRO at a fixed relative position. A small maneuver is commanded to begin the v-bar approach (blue portion of the trajectory), and subsequent hops are performed until the SRO is near the OS. At this point, the green portion of the trajectory begins and the controller allows the SRO to coast until it reaches the point of closest approach (the red x) at a range of 1.08 m, where another hold position maneuver is performed to represent OS capture. This final maneuver is for illustration purposes only and may not be necessary to capture the OS dynamically. Detailed capture dynamics are not simulated.

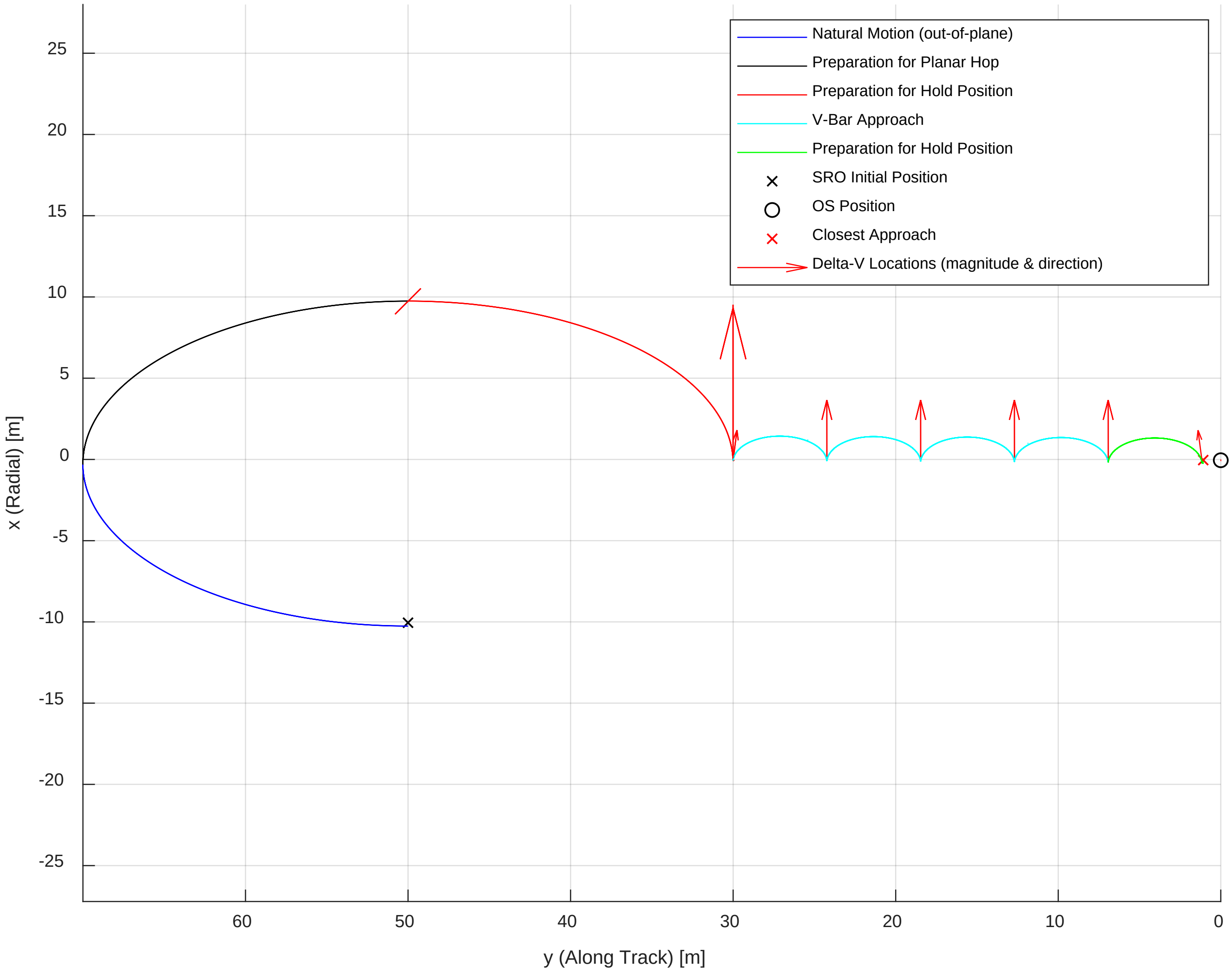


Figure 53 – Relative orbit *xy*-projection (LVLH) for nominal trajectory

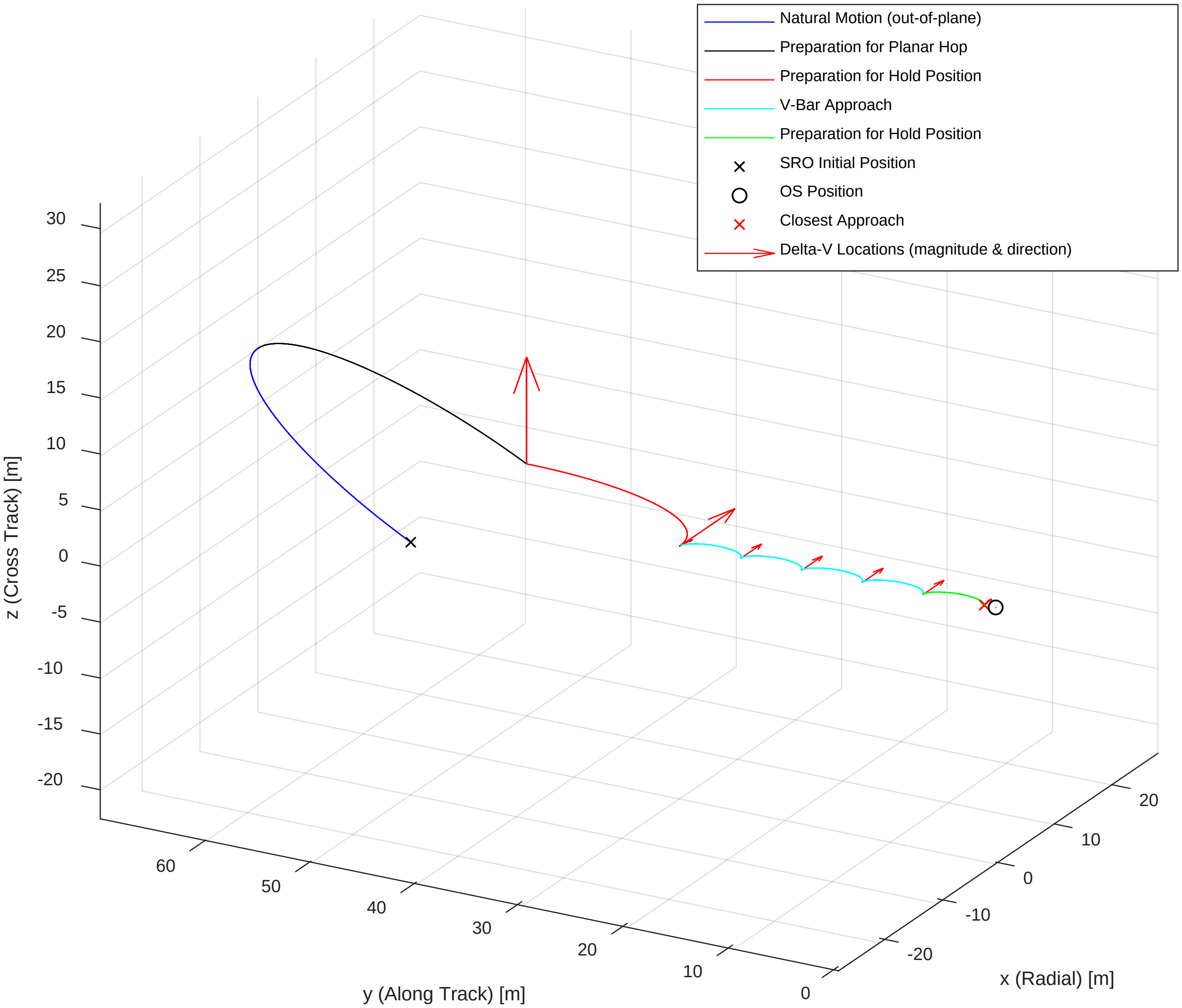


Figure 54 – Relative orbit three dimensional view (LVLH) for nominal trajectory

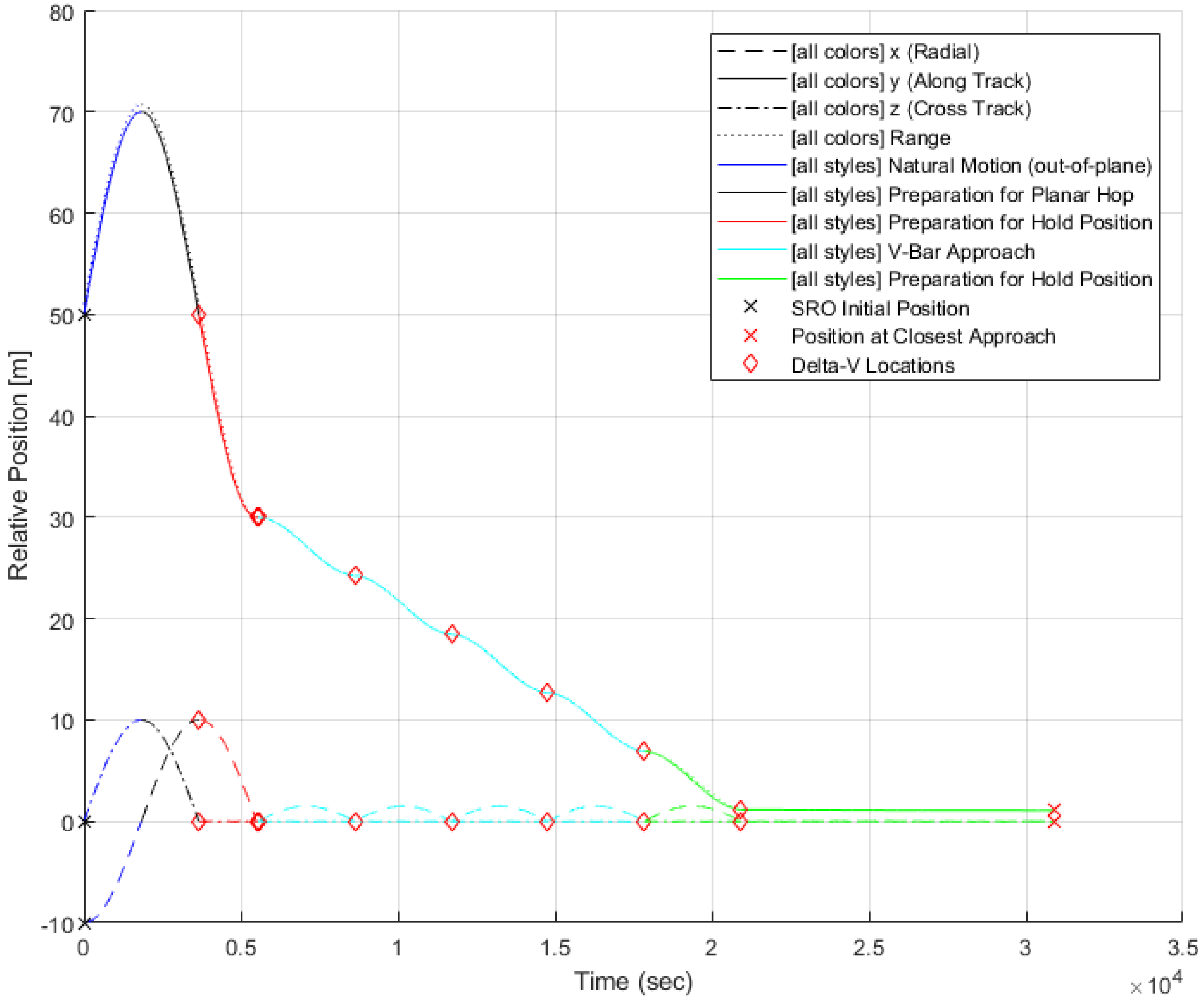


Figure 55 – Relative position time history (LVLH) for nominal trajectory

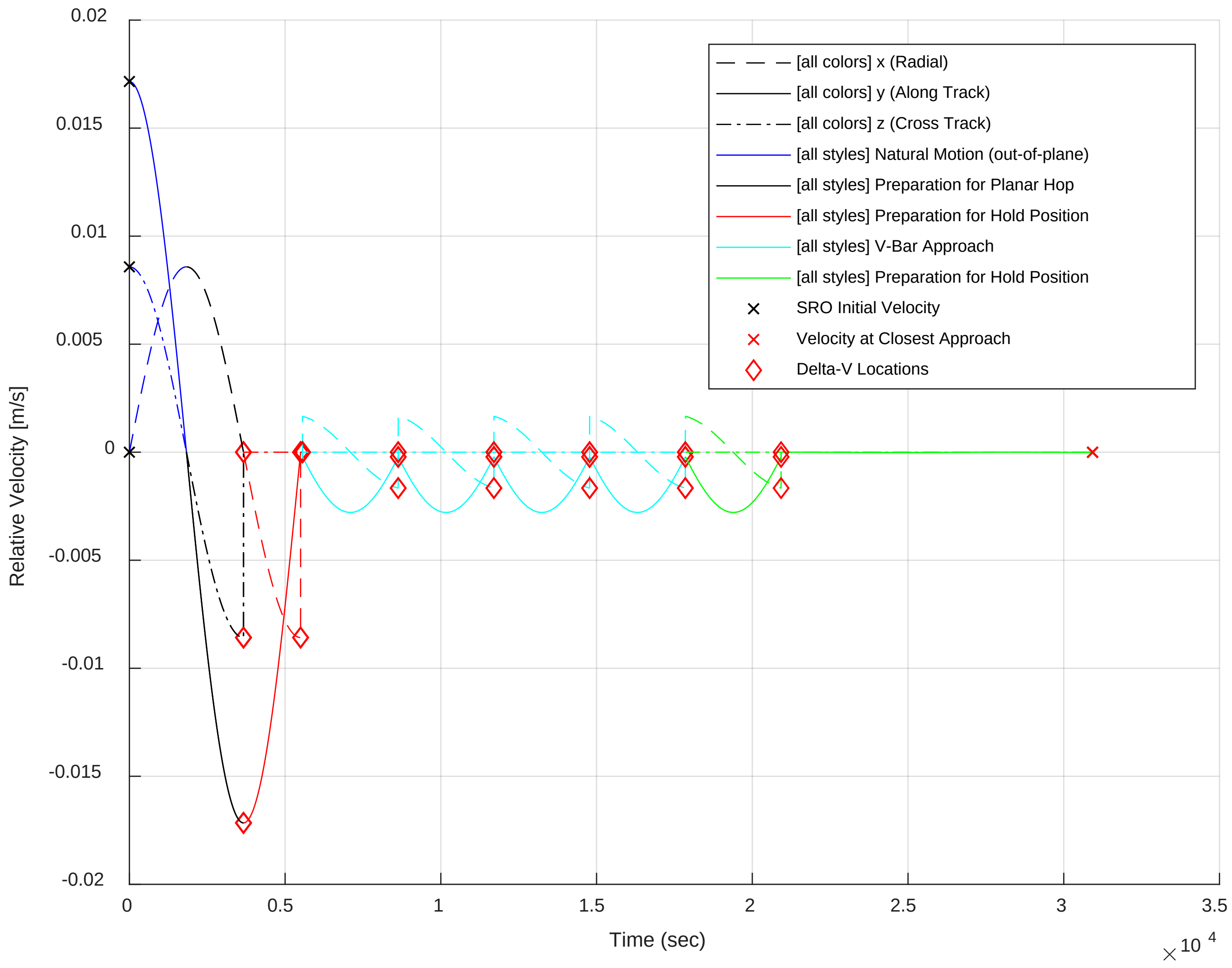


Figure 56 – Relative velocity time history (LVLH) for nominal trajectory

4.4.4.2 Passive Abort Trajectory

The passive abort trajectory is shown in Figures 57-60. This scenario is identical to the nominal approach until the passive abort occurs in the green portion of the trajectory. Assuming a fault has been detected, no additional maneuvers are performed, and the SRO passes through the point of closest approach at 1.15 m and does not perform a hold position maneuver. It simply continues in natural motion, which causes it to drift back away from the OS in the negative along-track direction until it reaches the standby distance of 50 m.

At this point, a safety ellipse maneuver is performed (magenta portion of the trajectory) to inject out-of-plane cross-track motion. Finally, a freeze drift maneuver is performed in the yellow portion of the trajectory to eliminate along-track drift of the out-of-plane passively safe standby ellipse.

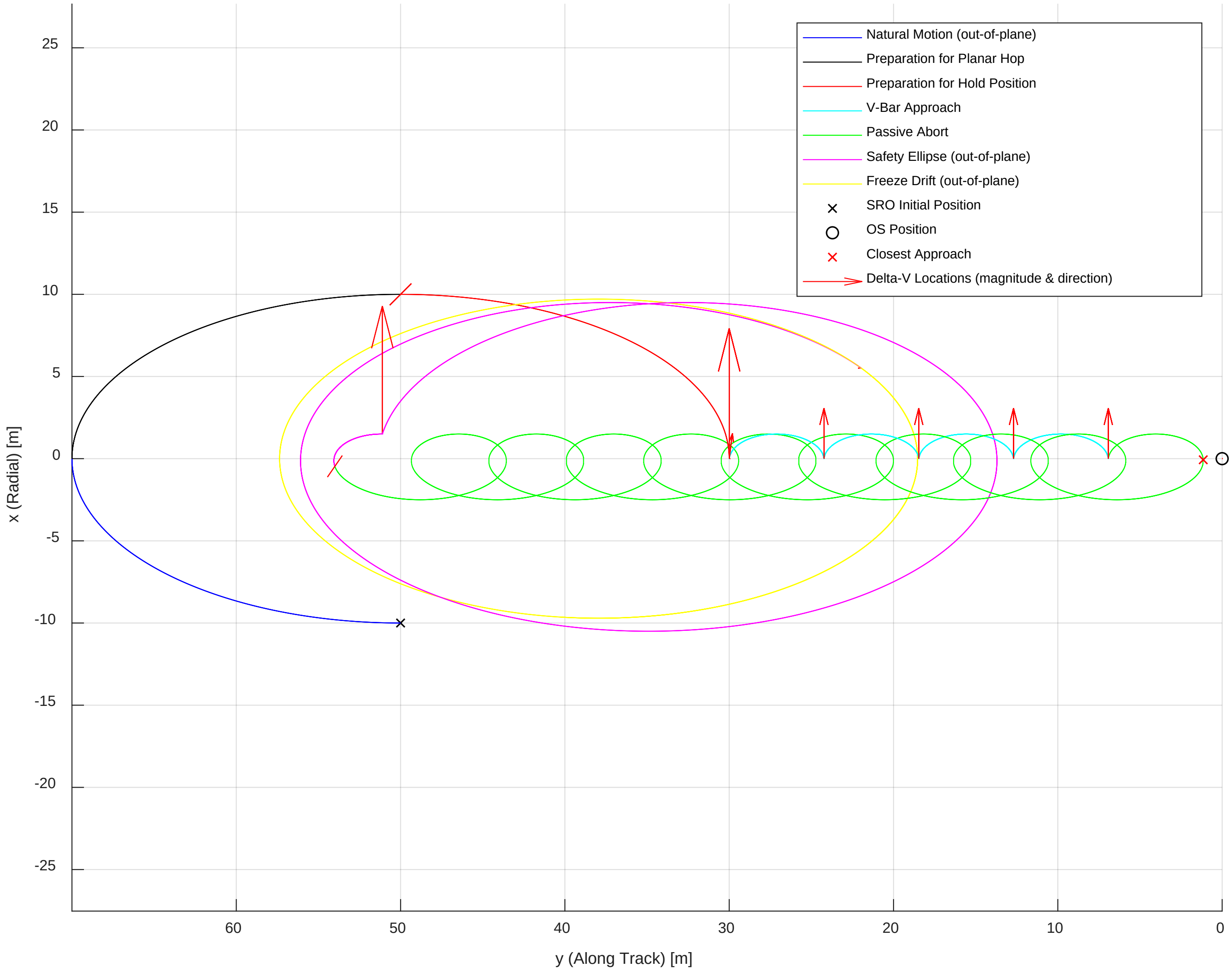


Figure 57 – Relative orbit *xy*-projection (LVLH) for passive abort

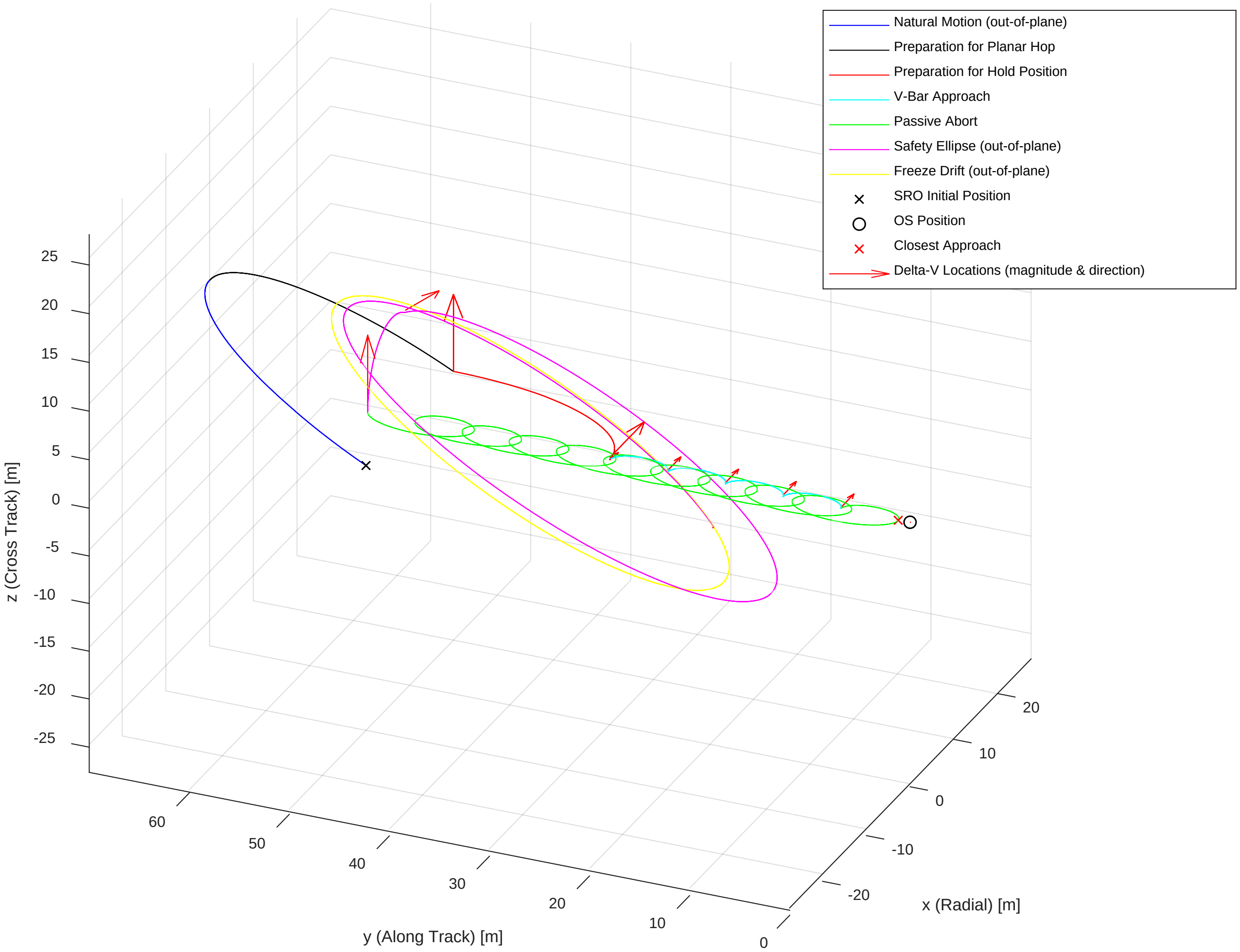


Figure 58 – Relative orbit three dimensional view (LVLH) for passive abort

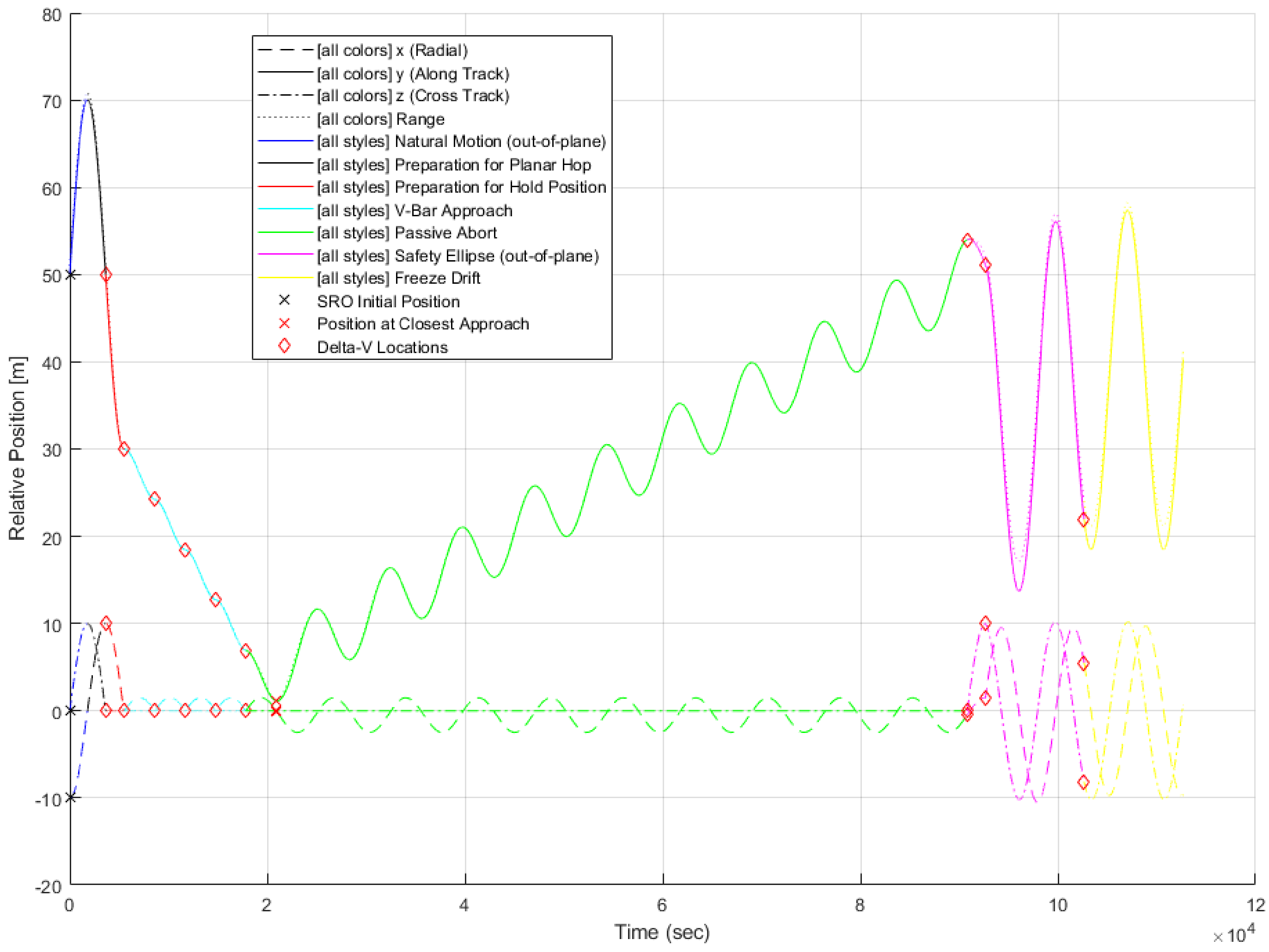


Figure 59 – Relative position time history (LVLH) for passive abort

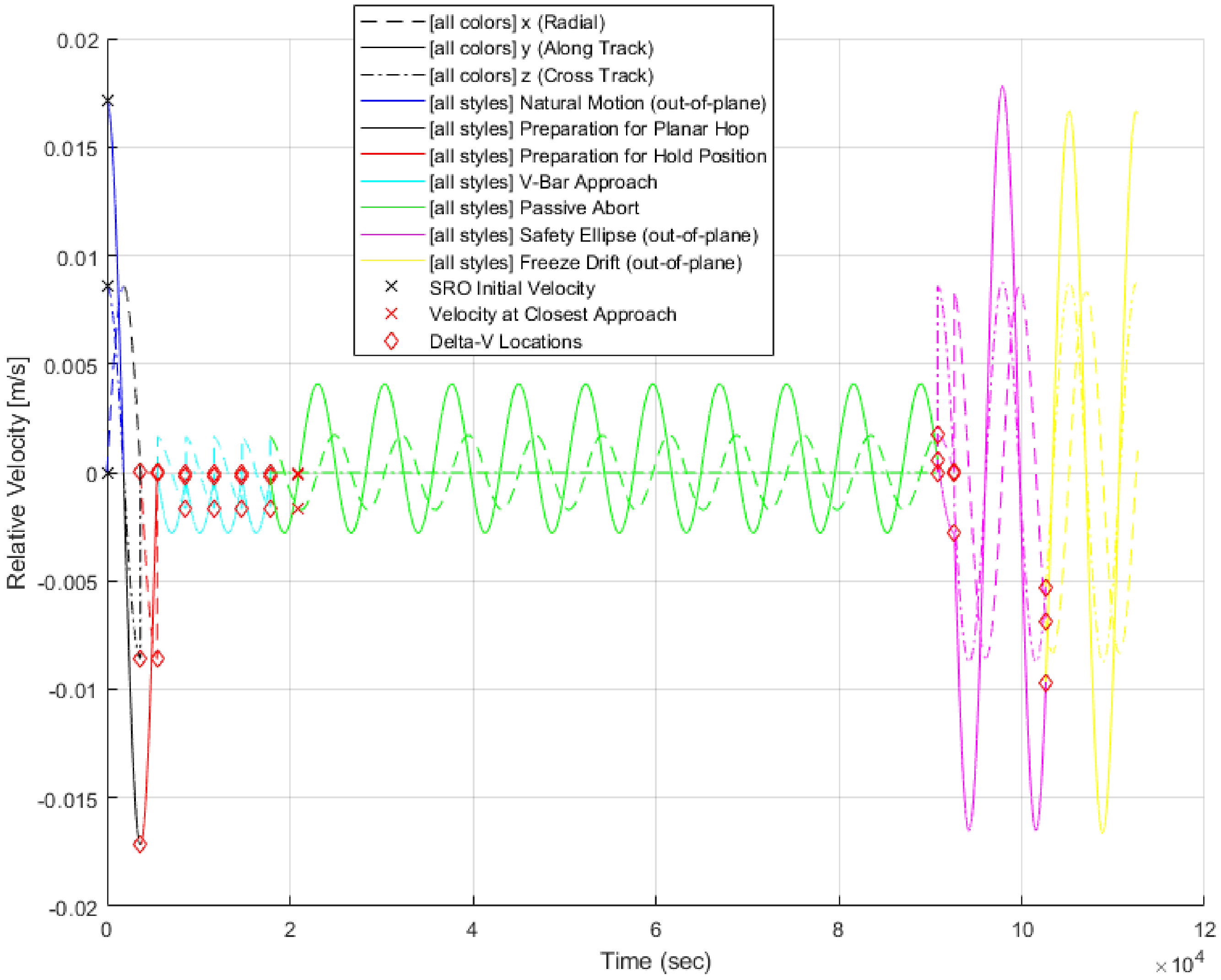


Figure 60 – Relative velocity time history (LVLH) for passive abort

4.4.4.3 Active Abort Trajectory

The active abort trajectory is shown in Figures 61-65. Figure 63 shows an additional 3D view to more clearly illustrate the active abort. This scenario is identical to the nominal approach until the active abort occurs in the green portion of the trajectory. Assuming a fault has been detected, a maneuver is performed to inject out-of-plane cross-track motion. This location becomes the point of closest approach at a range of 6.92 m. During the magenta portion of the trajectory, natural motion causes the out-of-plane ellipse to drift back away from the target in the negative along-track direction until it reaches the standby distance of 50 m from the OS. Finally, a freeze drift maneuver is performed in the yellow portion of the trajectory to eliminate along-track drift of the standby ellipse.

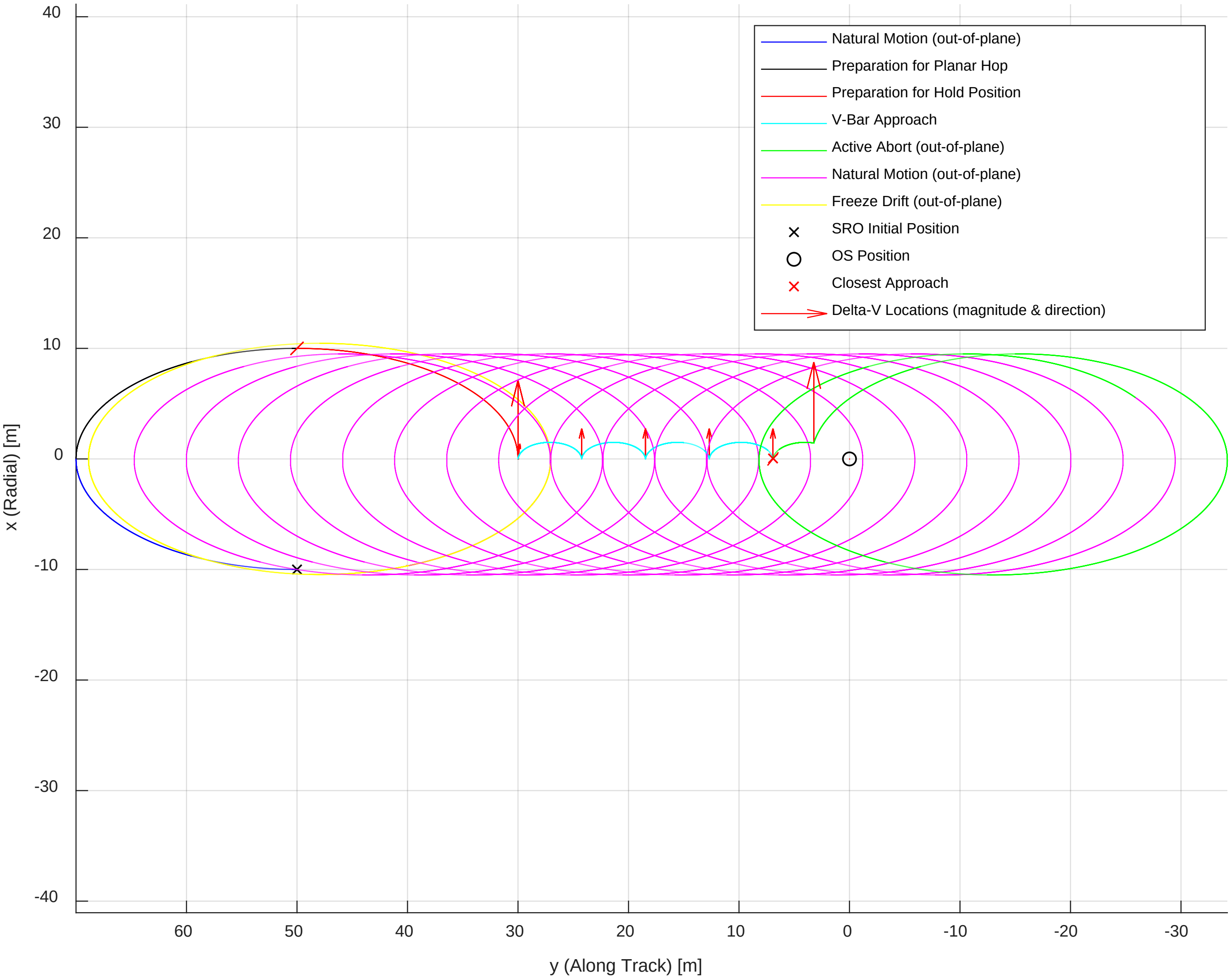


Figure 61 – Relative orbit *xy*-projection (LVLH) for active abort

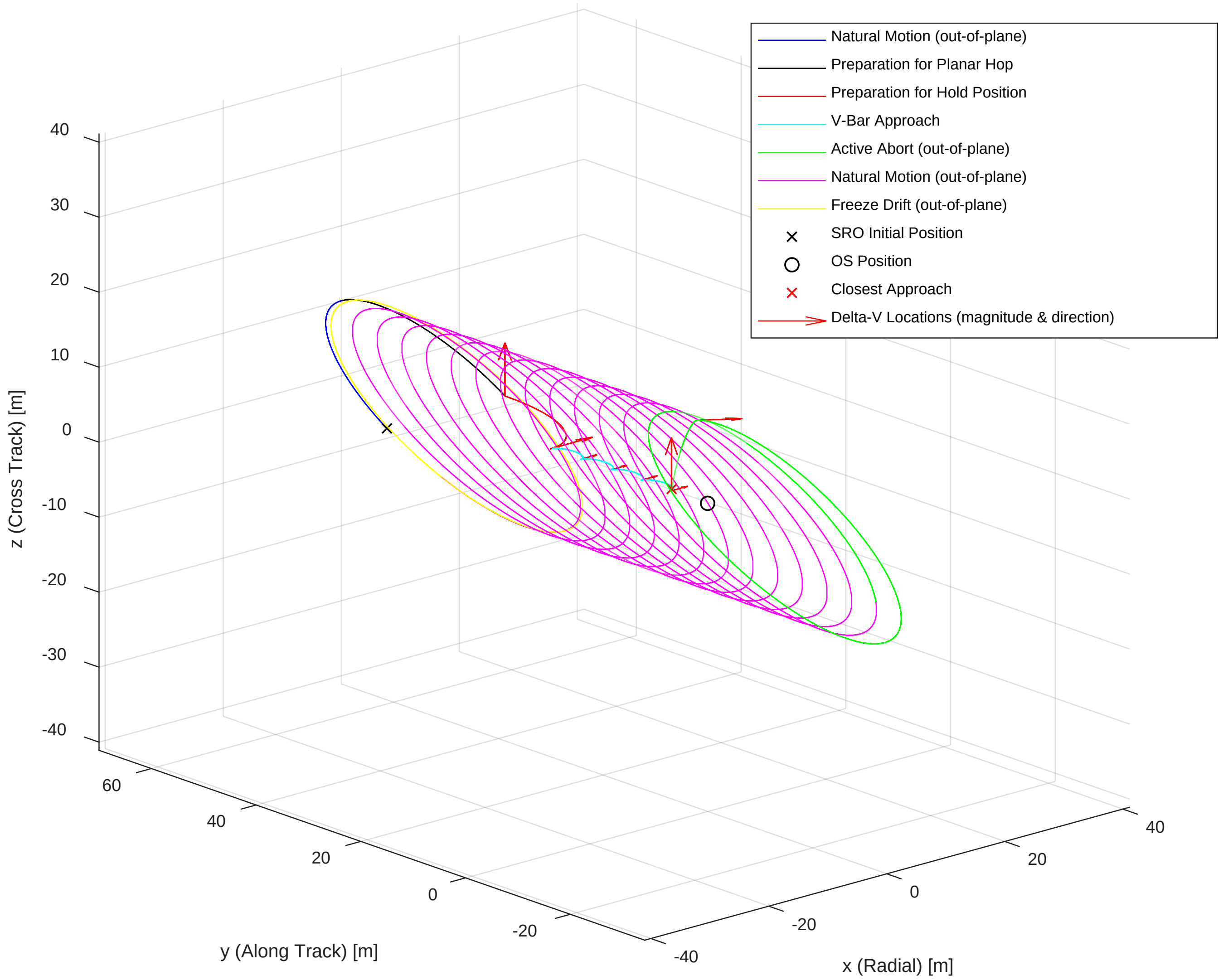


Figure 62 – Relative orbit three dimensional view (LVLH) for active abort

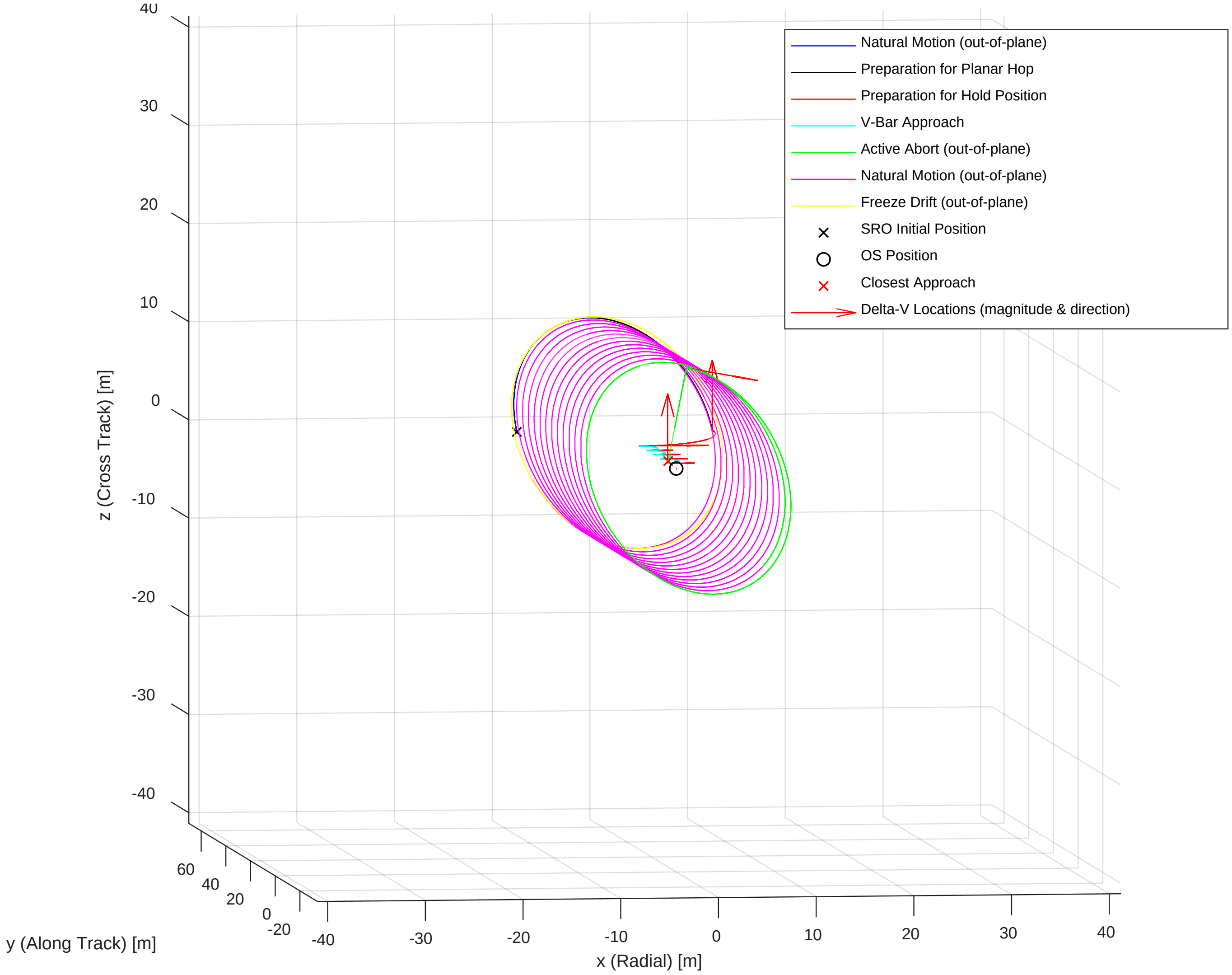


Figure 63 – Relative orbit three dimensional view (LVLH) for active abort

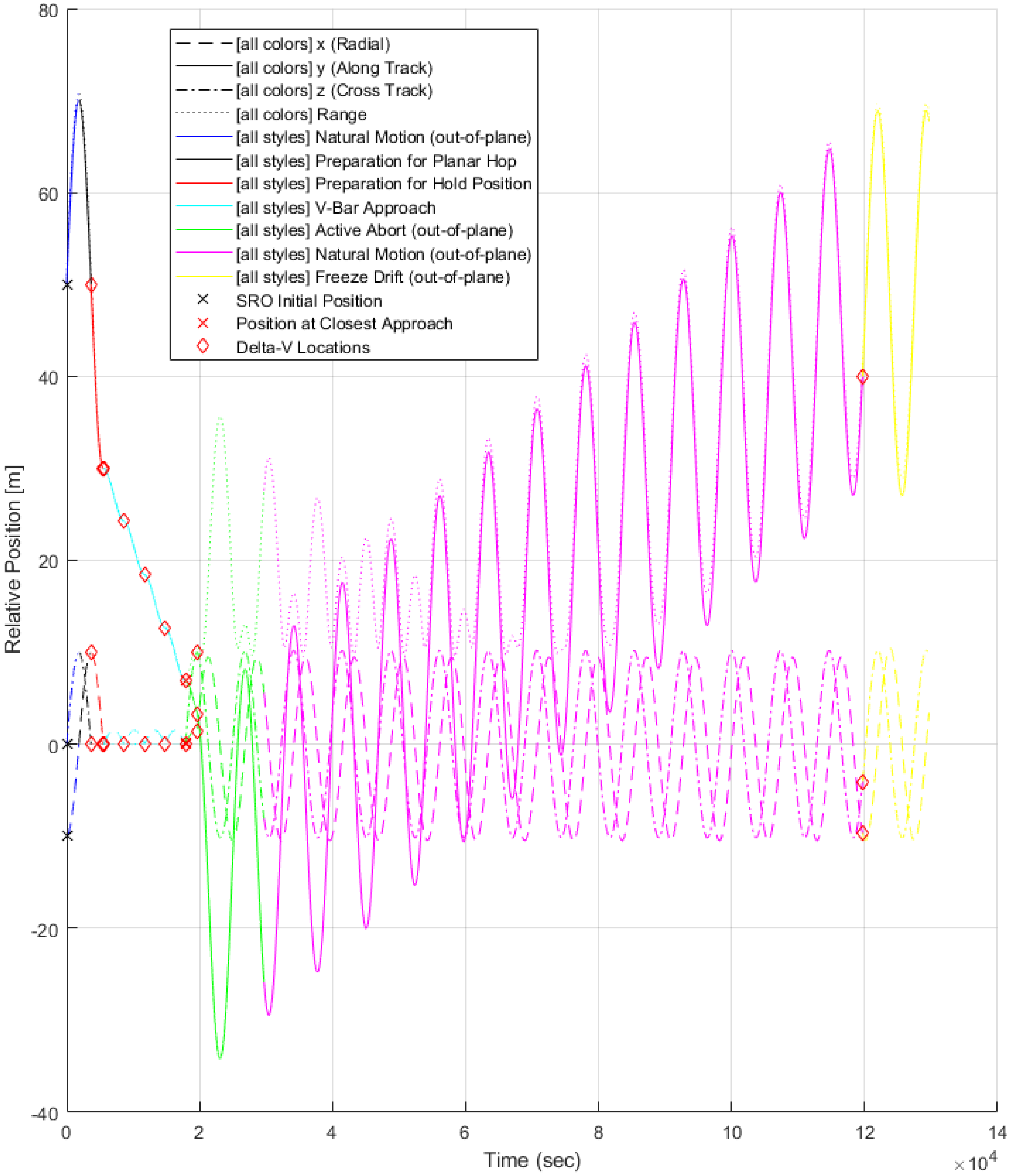


Figure 64 – Relative position time history (LVLH) for active abort

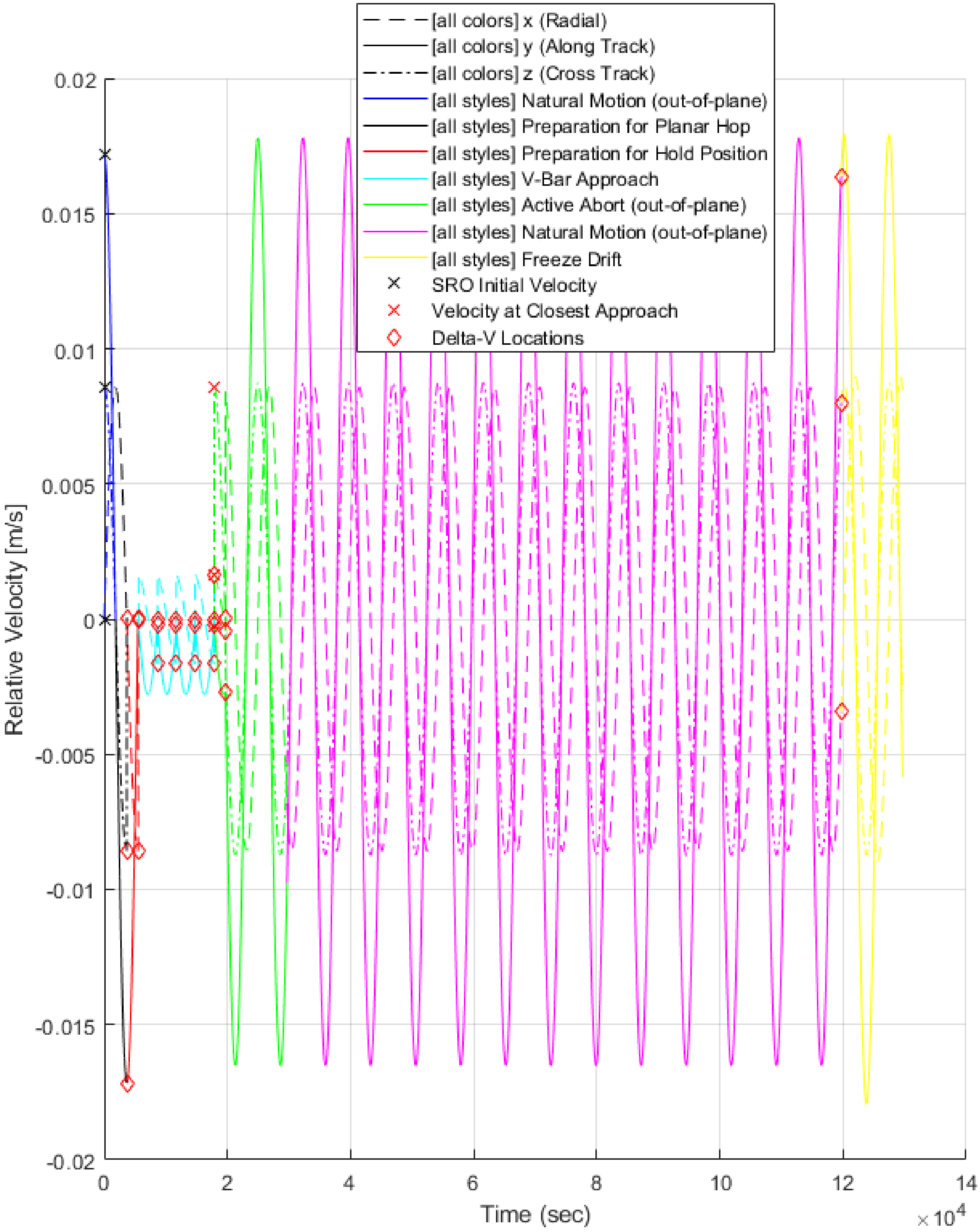


Figure 65 – Relative velocity time history (LVLH) for active abort

4.5 MSR Fault Protection Requirements Development

A desired set of fault protection behaviors is established in this section for the autonomous rendezvous and capture phase of the SRO mission. A fault tree analysis is performed to determine which faults should be considered based on input from subject matter experts. Several major areas are considered, including relative orbit determination, guidance & control, sequencing, and capture operations. Next, a selected subset of faults in each of these areas is expanded in detail. Criticality, detection, diagnosis, and response strategies are examined at various stages of the rendezvous and capture process. These details are used to define a set of potential fault protection requirements that accounts for different conditions in different stages of the process. In the following section, a fault protection architecture is developed that shows how fault protection could be implemented during this unique and challenging mission phase.

The terminal rendezvous and OS capture scenario provides an excellent case study for fault protection research. The initial portion of this study develops a desired set of FP behaviors for autonomous rendezvous and capture of the OS. This has been done in collaboration with the Mars Sample Return (MSR) mission formulation team at JPL. The MSR rendezvous working group is made up of members from three disciplines: relative orbit determination, guidance & control, and sequencing. A separate ROCS team is developing concepts for the flight hardware subsystem that will perform the capture operation. Inputs have also been sought from the SRO flight systems working group about various aspects of spacecraft subsystem concepts.

There are three desired outcomes for the MSR team. First, a set of initial FP requirements should be defined. These requirements may then be used to drive the initial design of the SRO rendezvous and capture system. Next, the MSR team desires to integrate fault protection with mission concept development, influencing design decisions during Pre-Phase A based on FP considerations. Finally, the MSR team desires to apply the FP process used in this study to other aspects of MSR mission design.

Several key requirements guided this study. First, mission success is vital. Fault protection should be designed to ensure the SRO mission to capture the OS can be completed or aborted without ground intervention, even under fault conditions. As stated earlier, autonomy is a key feature, since terminal rendezvous and capture occur fully autonomously. Safety is also a key concern, and fault protection should prevent the spacecraft from colliding with the OS. Finally, time criticality should be taken into account. For example, a fault response may be quite different at the beginning of the autonomous rendezvous sequence when the SRO is 100 m from the OS than in the last 5 or 10 meters.

Several different tasks were undertaken in order to define the fault protection behavior for autonomous rendezvous and capture. Some of these are standard fault protection practices, and others were customized for this study. All tasks have been completed at the preliminary level only, since detailed design has not yet begun for this mission concept.

4.5.1 MSR GN&C Subsystem Taxonomy and Fault Trees

To aid in clarifying terminology, a subsystem taxonomy (or system block diagram), shown in Figure 66, was constructed based on the generic subsystem taxonomy. This diagram lists out various elements, subsystems, and components of the system and was also used to help team members understand conceptually what components should be considered for the fault protection process. An example of terminology clarification is the naming of various rendezvous cameras, shown in the expansion of Figure 66. Because the terms “Narrow Angle Camera” and “Wide Angle Camera” have different meanings in different contexts, the rendezvous team developed animal names for each camera. The “hawk” is a camera that can see far away, the “dog” is a shorter-range camera with a wider field of view, and the “fish” is a very wide-angle camera with a short range. A readable version of the complete Subsystem Taxonomy is shown in Appendix C.

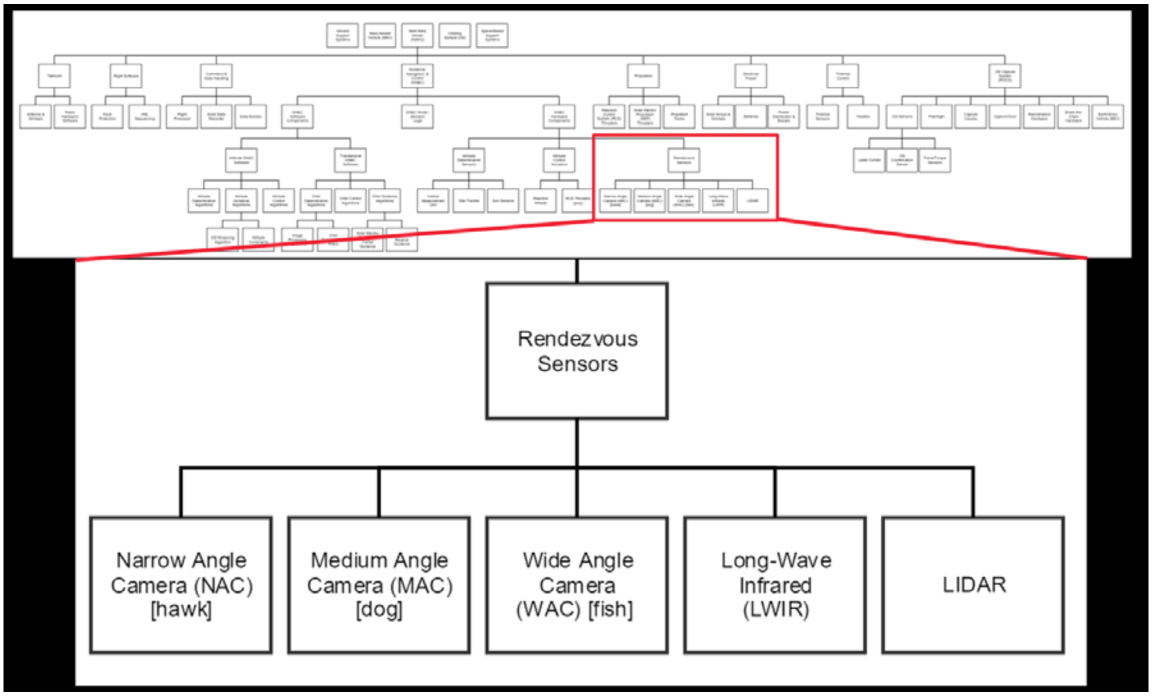

Rendezvous Sensors
Narrow Angle Camera (NAC) [hawk]
Medium Angle Camera (MAC) [dog]
Wide Angle Camera (WAC) [fish]
Long-Wave Infrared (LWIR)
LIDAR

Figure 66 – Subsystem taxonomy, with an example expanded [53]

An important step in developing fault protection requirements is to perform a fault tree analysis. Through discussions with subject-matter experts from the MSR rendezvous working group, a fault tree was defined that captures faults that could result in failure of the terminal rendezvous phase, as shown in Figure 67.

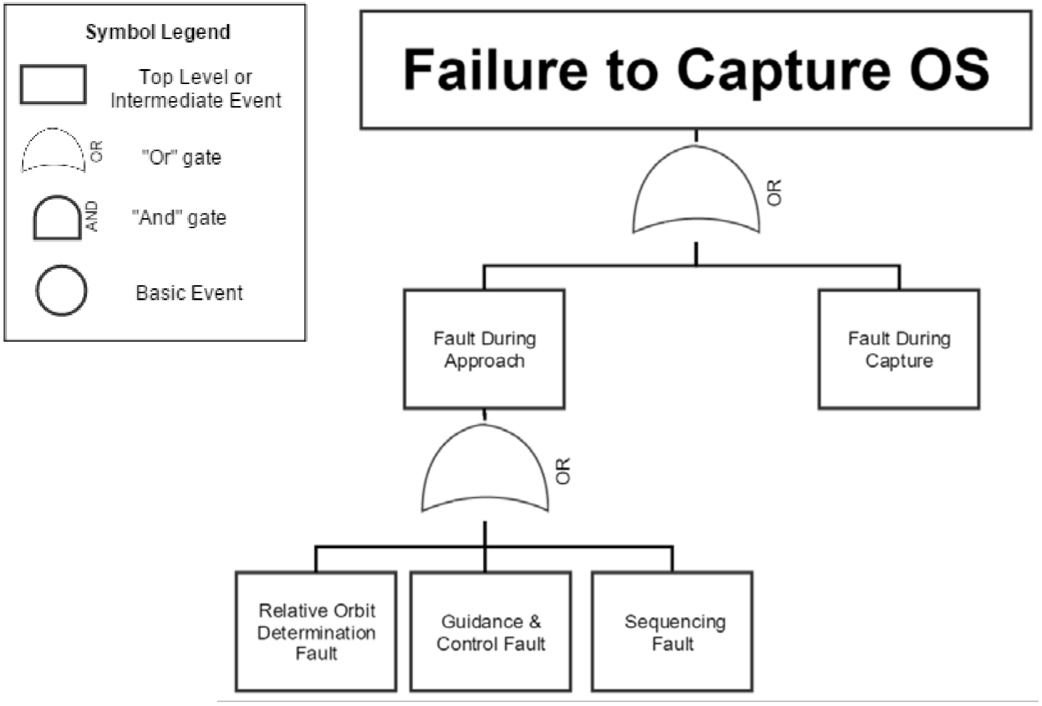


Figure 67 – High-level fault tree for autonomous rendezvous & capture [53]

The first discipline considered is Relative Orbit Determination, which involves calculating relative position & velocity from rendezvous sensor data. Next is Guidance & Control, which is responsible for attitude determination/control and trajectory control during rendezvous. In addition, sequencing uses the Virtual Machine Language (VML) to direct the autonomous process based on state machines that are developed on the ground and loaded onboard [72]. Finally, Capture deals with the mechanical and logical components for capturing the OS.

An initial fault tree was developed prior to consulting the MSR team. However, in order to capture the inputs from JPL experts representing each discipline, various breakout meetings were held to revise and expand this initial fault tree. These meetings were designed to simply brainstorm, add, remove, rearrange, or rename potential faults from the fault tree. Figure 68 shows one example of the results of these breakout sessions.

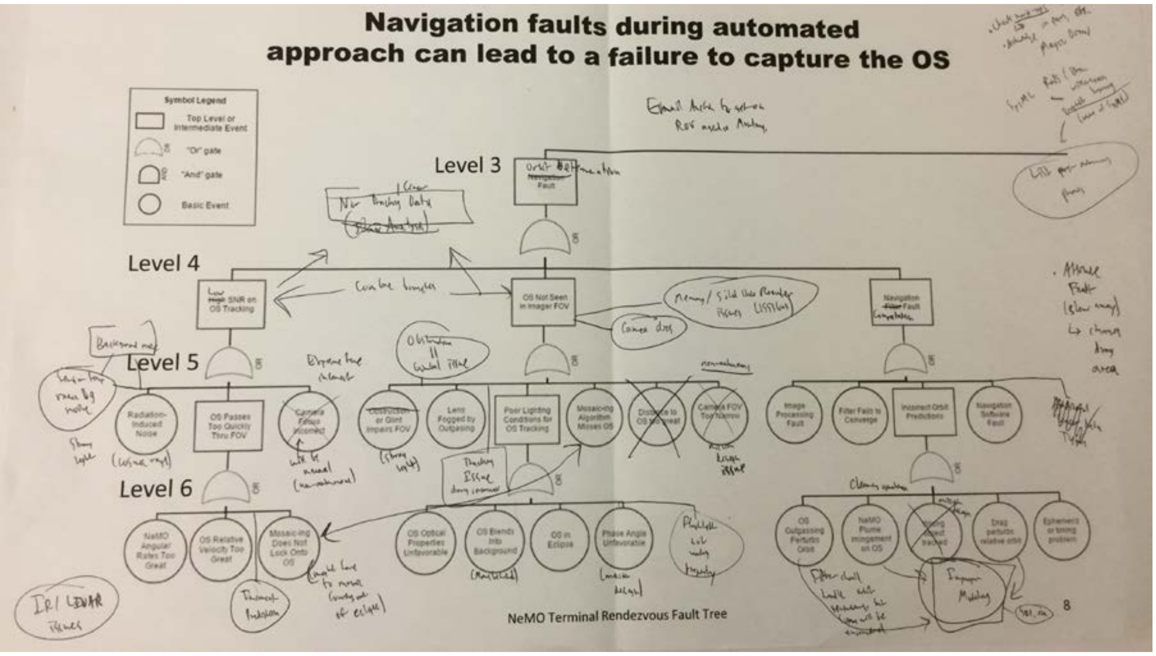

Navigation faults during automated approach can lead to a failure to capture the OS
Symbol Legend
Top Level or Intermediate Event
"Or" gate
"And" gate
Basic Event
Level 3
Orbit Determination
~~Navigation~~ Fault
Level 4
Low ~~High~~ SNR on OS Tracking
OS Not Seen in Imager FOV
Navigation ~~Filter~~ Fault
Computation
Combine branches
Camera dies
No Tracking Data
Level 5
Background noise
Radiation-Induced Noise
OS Passes Too Quickly Thru FOV
~~Camera Focus Incorrect~~
Obstruction or Glint Impairs FOV
Lens Fogged by Outgassing
Poor Lighting Conditions for OS Tracking
Mosaicing Algorithm Misses OS
~~Distance to OS too great~~
~~Camera FOV Too Narrow~~
Image Processing Fault
Filter Fails to Converge
Incorrect Orbit Predictions
Navigation Software Fault
Stray light
Tracking Issue
Level 6
NeMO Angular Rates Too Great
OS Relative Velocity Too Great
Mosaicing Does Not Lock Onto OS
OS Optical Properties Unfavorable
OS Blends Into Background
OS in Eclipse
Phase Angle Unfavorable
OS Outgassing Perturbs Orbit
NeMO Plume Impingement on OS
~~Wrong object tracked~~
Drag perturbs relative orbit
Ephemeris or timing problem
Attitude Fault
IR / LIDAR
NeMO Terminal Rendezvous Fault Tree
8

Figure 68 – Result of a fault tree brainstorming session
for relative orbit determination [53]

Finally, the results of all the breakout discussions were compiled to create comprehensive fault trees. A complete fault tree including over 50 root cause events (shown in Appendix D in text form) was constructed to capture all faults specific to rendezvous and capture. A second fault tree shown in

Figure 69 was used to capture generic spacecraft subsystem (non-GN&C) faults that could occur during rendezvous and capture.

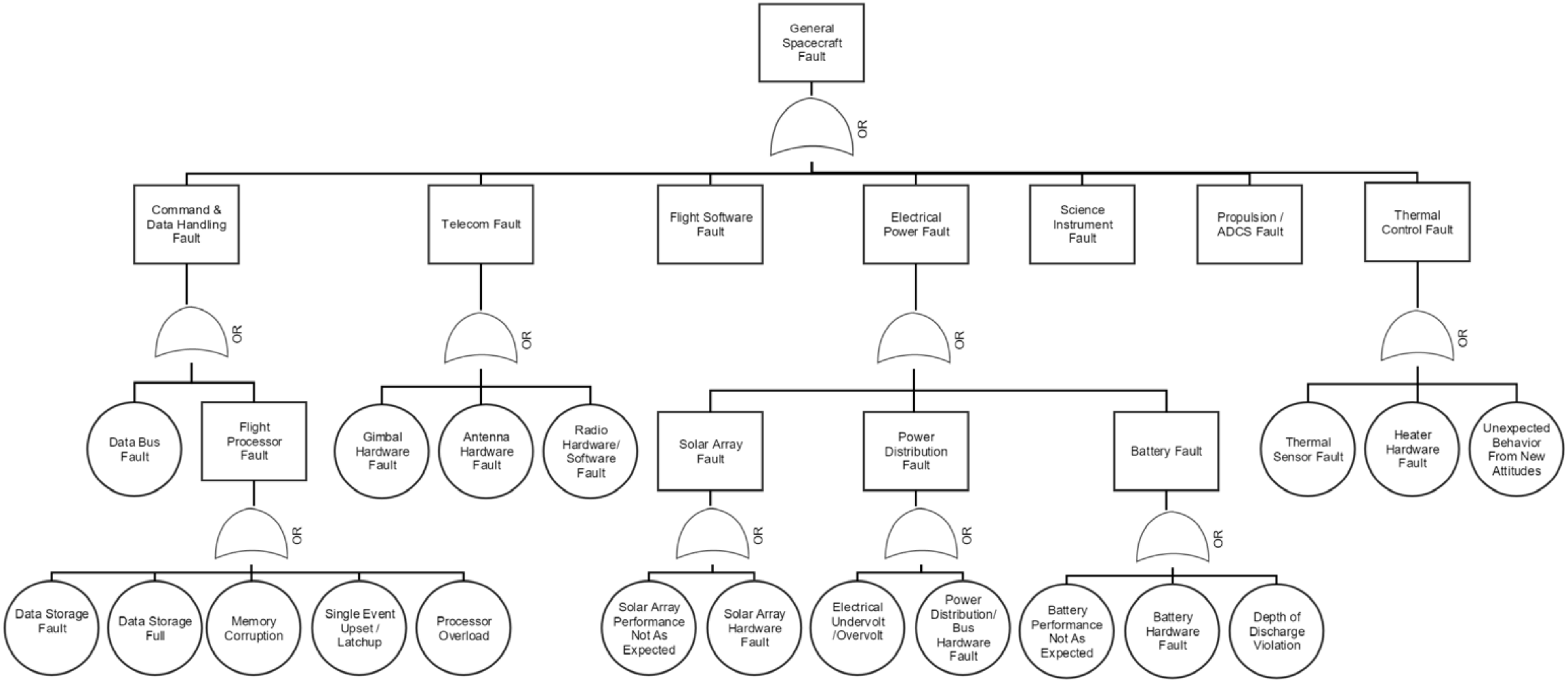
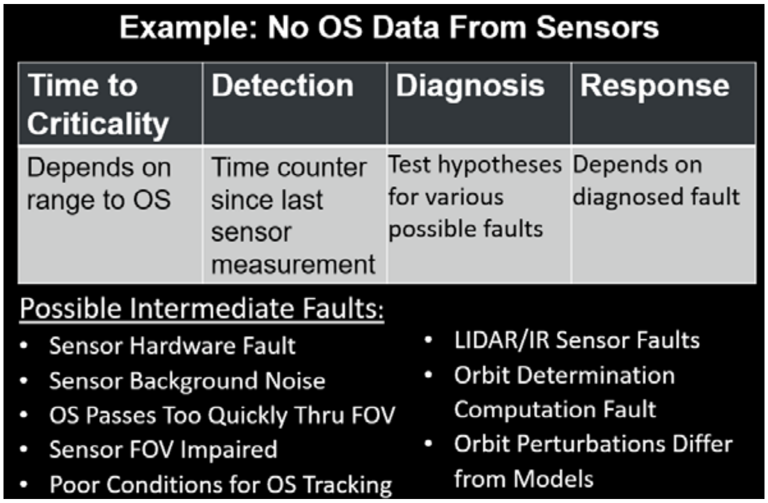


Figure 69 – General spacecraft subsystem fault tree [53]

4.5.2 MSR Rendezvous & Capture Requirements

A subset of faults (bolded in the fault tree shown in Appendix D) was selected from the completed fault tree. Several representative faults were chosen from each discipline (relative orbit determination, guidance & control, sequencing, and capture). The selected faults are challenging to detect, diagnose, or respond to in a quick, efficient, and safe way. These faults were expanded in detail, and time-to-criticality, detection methods, diagnosis methods, and response strategies are examined at various stages of the rendezvous and capture process. These details were then used to define a set of potential fault protection requirements that accounts for different conditions in different stages of the rendezvous process. One example strategy for a single fault is shown in Figure 70, and the related possible fault protection requirements are shown in Figure 71. Note that both the time to criticality and response strategy for this particular fault are too complex to fit into the table in Figure 70 and are described in depth in the following section. Details of strategies and requirements for all of the selected faults are provided in Appendix E. After completion of these requirements, a second round of breakout meetings was held with technical experts in each area to share the results and seek direction for the next steps.



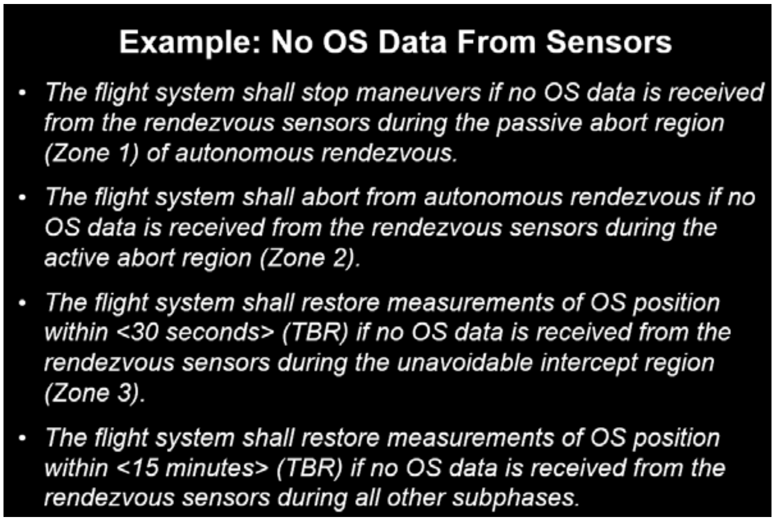
Example: No OS Data From Sensors

Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Time counter since last sensor measurement	Test hypotheses for various possible faults	Depends on diagnosed fault

Possible Intermediate Faults:

- Sensor Hardware Fault
- Sensor Background Noise
- OS Passes Too Quickly Thru FOV
- Sensor FOV Impaired
- Poor Conditions for OS Tracking
- LIDAR/IR Sensor Faults
- Orbit Determination Computation Fault
- Orbit Perturbations Differ from Models

Figure 70 – Example fault protection strategy [53]



Example: No OS Data From Sensors

- *The flight system shall stop maneuvers if no OS data is received from the rendezvous sensors during the passive abort region (Zone 1) of autonomous rendezvous.*
- *The flight system shall abort from autonomous rendezvous if no OS data is received from the rendezvous sensors during the active abort region (Zone 2).*
- *The flight system shall restore measurements of OS position within <30 seconds> (TBR) if no OS data is received from the rendezvous sensors during the unavoidable intercept region (Zone 3).*
- *The flight system shall restore measurements of OS position within <15 minutes> (TBR) if no OS data is received from the rendezvous sensors during all other subphases.*

Figure 71 – Example of possible fault protection requirements [53]

4.6 Fault Detection, Diagnosis, and Recovery Strategy

An example case (“No OS data received from sensors”) was selected to demonstrate how a spacecraft fault protection system may diagnose faults on-board. The fault protection architecture utilizes state machines so that fault responses are tied to the state of the system rather than simply as a reaction to the detection of symptoms.

The study focuses on four faults as a proof of concept. Each of these faults has a very similar symptom used for initial detection, but each fault also has a distinct diagnosis and response procedure. The four faults are “Sensor Loses Power”, “SRO Angular Rates Too Great”, “OS in Eclipse” and “Filter Does Not Converge”. Initial fault detection is performed by examining the result of an image processing algorithm developed for the Prox-1 small satellite mission [73]. This algorithm receives the simulated sensor image of the OS as an input and outputs a simple logical variable called `InView.` This variable is set to 1 if the OS is visible in the image and to 0 if the OS is not visible. An image timer tracks the amount of time since the last OS image has been seen, and if this timer surpasses a user-defined time threshold (and the OS has been seen previously), then a fault detection is triggered. A fault detection can also be triggered by an input variable indicating relative orbit filter non-convergence.

Once a fault is detected, the diagnostic state machine shown in Figure 72 is activated. This state machines operates exactly like the generic diagnostic state machine described in Section 2.6. The diagnostic state machine consists of two primary states: NoFaultDetected and FaultDetected. During all nominal mission phases, NoFaultDetected is activated, but when one of the fault detection triggers described above is observed, FaultDetected will be activated. If the rendezvous process state machine in Figure 49 has any state active other than the Passive Miss or Active Abort Regions (Zones 1 and 2), then the diagnostic state machine enters the “Diagnose” state immediately when a fault is detected. If the rendezvous process is in the Passive Miss Region (`ZonePassive=1`) or Active Abort Region (`ZoneActive=1`), then the diagnostic state machine does not attempt to determine which fault has occurred. An active or passive abort maneuver is commanded

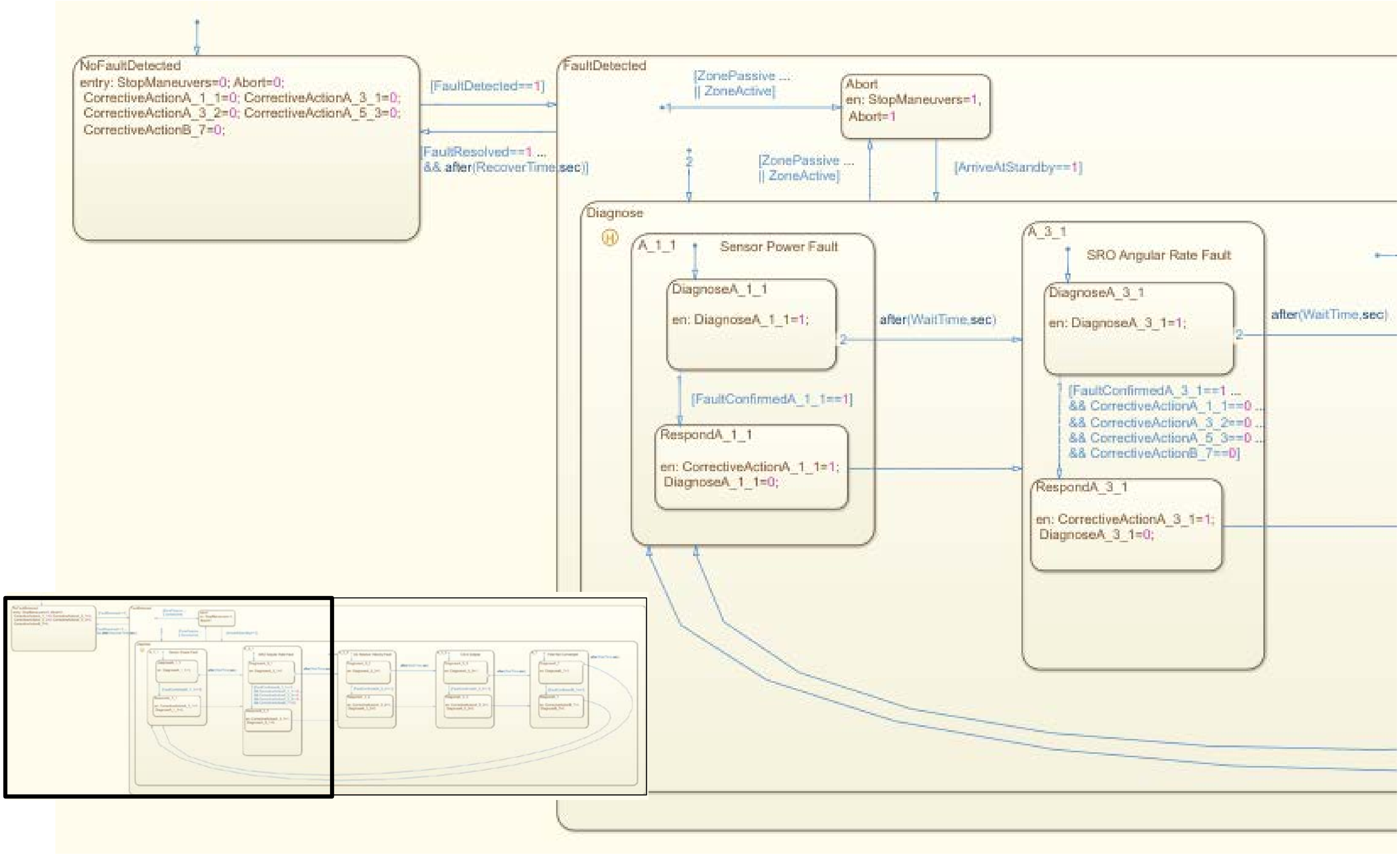


Figure 72 – Diagnostic state machine for MSR Simulation

immediately, returning the spacecraft to a passively safe standby orbit as described in Section 4.4.3 before entering the Diagnose state.

The Diagnose state consists of sub-states for each possible fault, which operate as described in Section 2.6. The “Sensor Loses Power” fault is diagnosed if `SensorPowerState` is equal to zero. “SRO Angular Rate Fault” is diagnosed if the maximum absolute value of any element of the angular velocity vector exceeds the user defined `MaxAllowableRate`. “OS in Eclipse” is diagnosed if `OSEclipseState` is equal to one. Finally, “Filter Does Not Converge” is diagnosed if `FilterConvergenceStatus` is equal to zero.

Once a fault is diagnosed, the diagnostic state machine calls the appropriate fault response. If “Sensor Loses Power” is diagnosed, the response is to send a command to reset the sensor power. If “SRO Angular Rates Too Great” is diagnosed, the response is to send a command to point the rendezvous sensor at the target. If “OS in Eclipse” is diagnosed, the response is to turn on a flashlight to enable imaging the OS during eclipse. Finally, if “Filter Does Not Converge” is diagnosed, the response is to reset the navigation filter.

4.7 Evaluation in Simulation

A simulation has been developed for the MSR application in MATLAB/Simulink to evaluate the generic FDIR architecture described in Chapter 2. This investigation builds on capabilities developed at Georgia Tech for other projects, including a high-fidelity ProxOps GN&C simulation for the Prox-1 small satellite mission [15,31], simulation models for feasibility studies of constellations of CubeSats at Mars [32], and a PITL testbed for high-fidelity testing of avionics boards for relative ProxOps called SoftSim6D [33].

4.7.1 Simulation Description

An overview of the simulation is shown in Figure 73 and the interaction of various components is illustrated by the architecture block diagram in Figure 74. The diagram is based on the generic architecture block diagram from Section 2.3.5. It is divided into the simulation environment representing the system under control (entity being controlled) and the control system (entity exercising the control) according to terminology defined in [5].

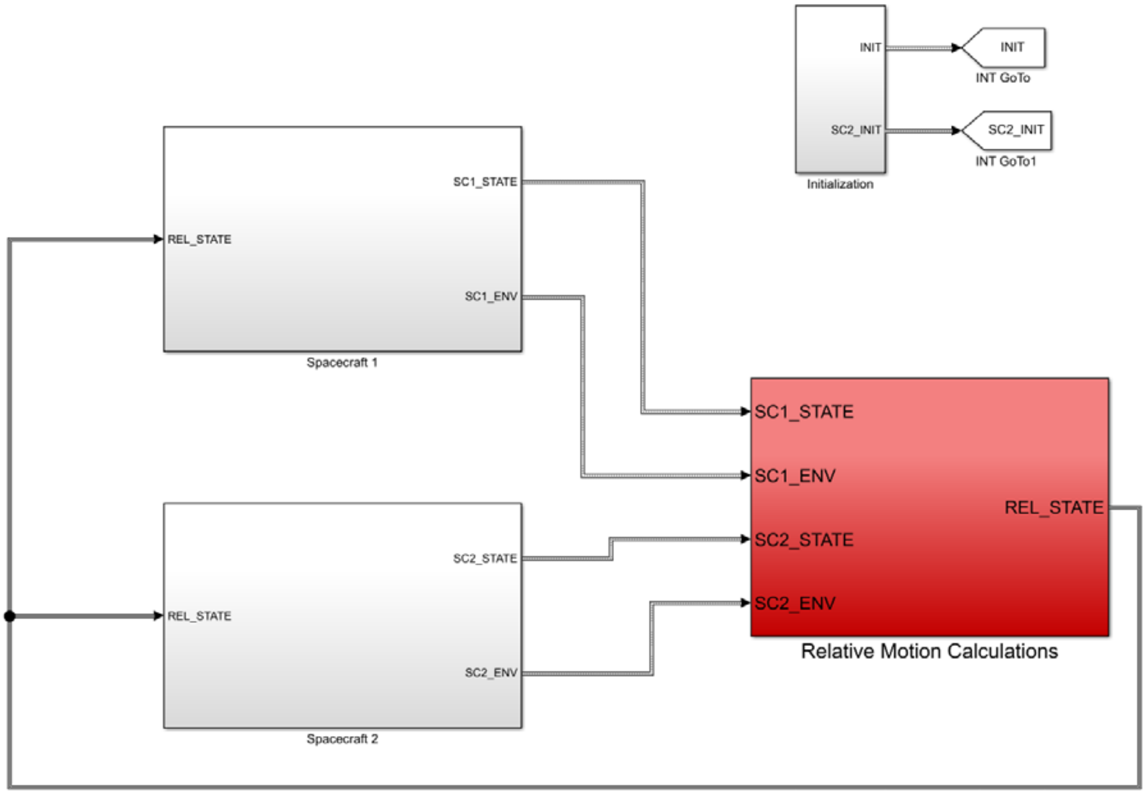


Figure 73 – Overview of MSR Simulation

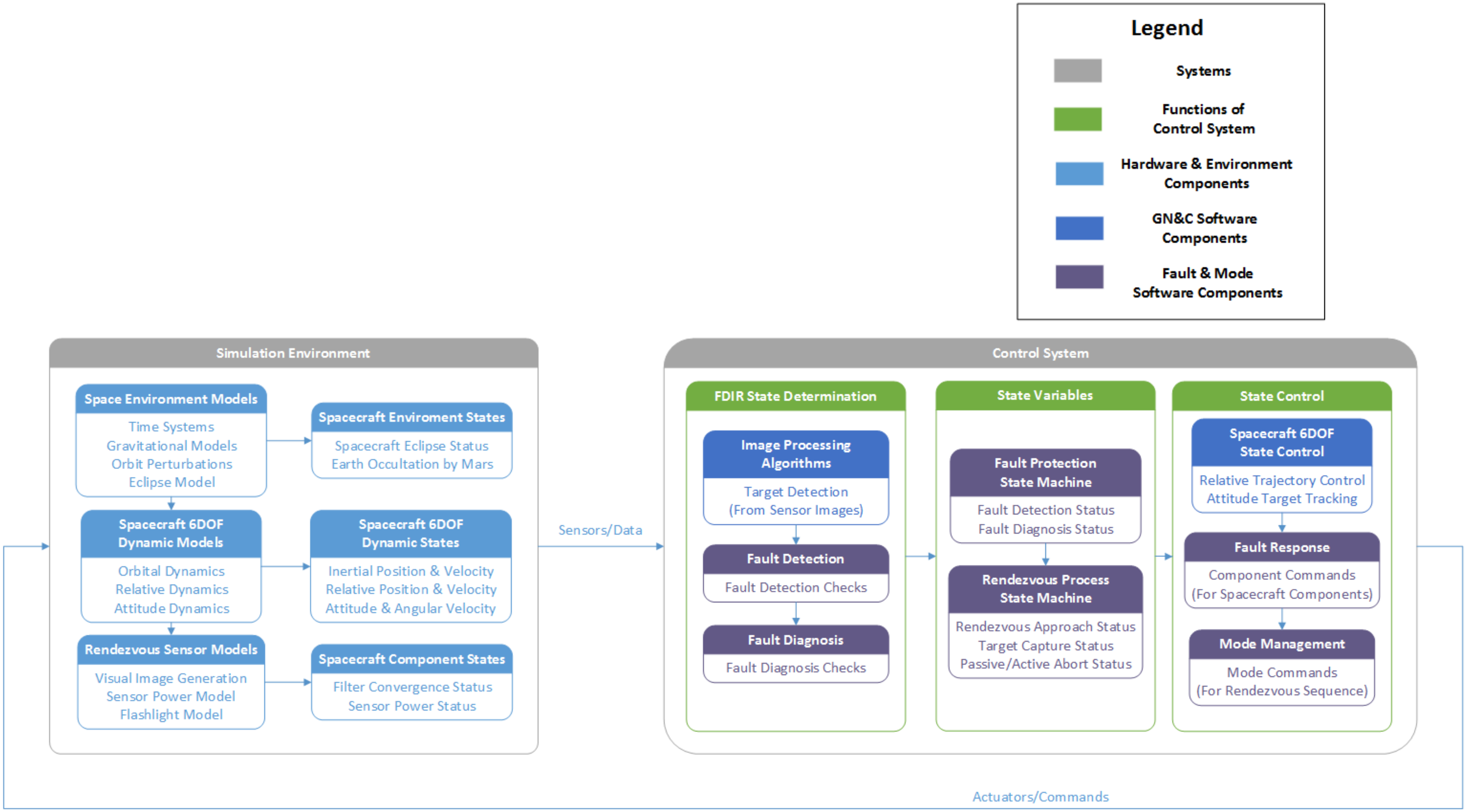

Legend
Systems
Functions of Control System
Hardware & Environment Components
GN&C Software Components
Fault & Mode Software Components
Simulation Environment
Space Environment Models
Time Systems
Gravitational Models
Orbit Perturbations
Eclipse Model
Spacecraft Enviroment States
Spacecraft Eclipse Status
Earth Occultation by Mars
Spacecraft 6DOF Dynamic Models
Orbital Dynamics
Relative Dynamics
Attitude Dynamics
Spacecraft 6DOF Dynamic States
Inertial Position & Velocity
Relative Position & Velocity
Attitude & Angular Velocity
Rendezvous Sensor Models
Visual Image Generation
Sensor Power Model
Flashlight Model
Spacecraft Component States
Filter Convergence Status
Sensor Power Status
Sensors/Data
Control System
FDIR State Determination
Image Processing Algorithms
Target Detection (From Sensor Images)
Fault Detection
Fault Detection Checks
Fault Diagnosis
Fault Diagnosis Checks
State Variables
Fault Protection State Machine
Fault Detection Status
Fault Diagnosis Status
Rendezvous Process State Machine
Rendezvous Approach Status
Target Capture Status
Passive/Active Abort Status
State Control
Spacecraft 6DOF State Control
Relative Trajectory Control
Attitude Target Tracking
Fault Response
Component Commands (For Spacecraft Components)
Mode Management
Mode Commands (For Rendezvous Sequence)
Actuators/Commands

Figure 74 – MSR Simulation Block Diagram

The simulation environment is composed of foundational SoftSim6D simulation components and state variables that track the status of these models. For example, space environment models include time systems, gravitational forces, and orbit perturbations. The initial simulation epoch is input as a Gregorian date & time and is then translated into a Julian Date. Force and moment perturbations are included for the SRO and OS orbits around Mars for J2 non-spherical gravity, third body effects from Phobos and Deimos, aerodynamic drag, solar radiation pressure, and gravity gradient. The ephemeris of the Sun in the Mars IAU frame is obtained from the JPL SPICE toolkit in order to determine when the OS and SRO are in eclipse. The ephemeris of the Earth in the Mars IAU frame is also obtained to determine when Earth is occulted by Mars and communication of the SRO with the ground is not possible.

The simulation environment also includes spacecraft dynamic models for inertial orbital dynamics, relative orbital dynamics, and attitude dynamics. The inertial orbit and attitude of each spacecraft is computed and numerically integrated separately, and the relative states are calculated by differencing the inertial states. A detailed model of the rendezvous visual sensor has also been developed. This model includes an image generation capability originally developed for Prox-1 [15]. The image generator takes the relative orbit and attitude of the SRO and OS as inputs to generate a simulated image of the OS as seen by the camera on the SRO. A screenshot of the Simulink sensor model (unreadable, but with descriptive labels) and a simulated image are shown in Figure 75.

In order for an image to be generated, the OS must be within the field of view (FOV) of the sensor, the sensor power must be turned on, and the OS must be in sunlight (not in eclipse). The sensor power model is a logical variable representing on/off states as 0 or 1.

The model features a fault injection capability which causes the sensor to lose power at a specified input time. Sensor power can be restored by sending a sensor reset command. There is also a flashlight model, which is a simple on/off state represented by 0 or 1. If the OS is in eclipse, the flashlight can be activated by a “flashlight on” command to illuminate it, allowing an image to be generated in the absence of sunlight.

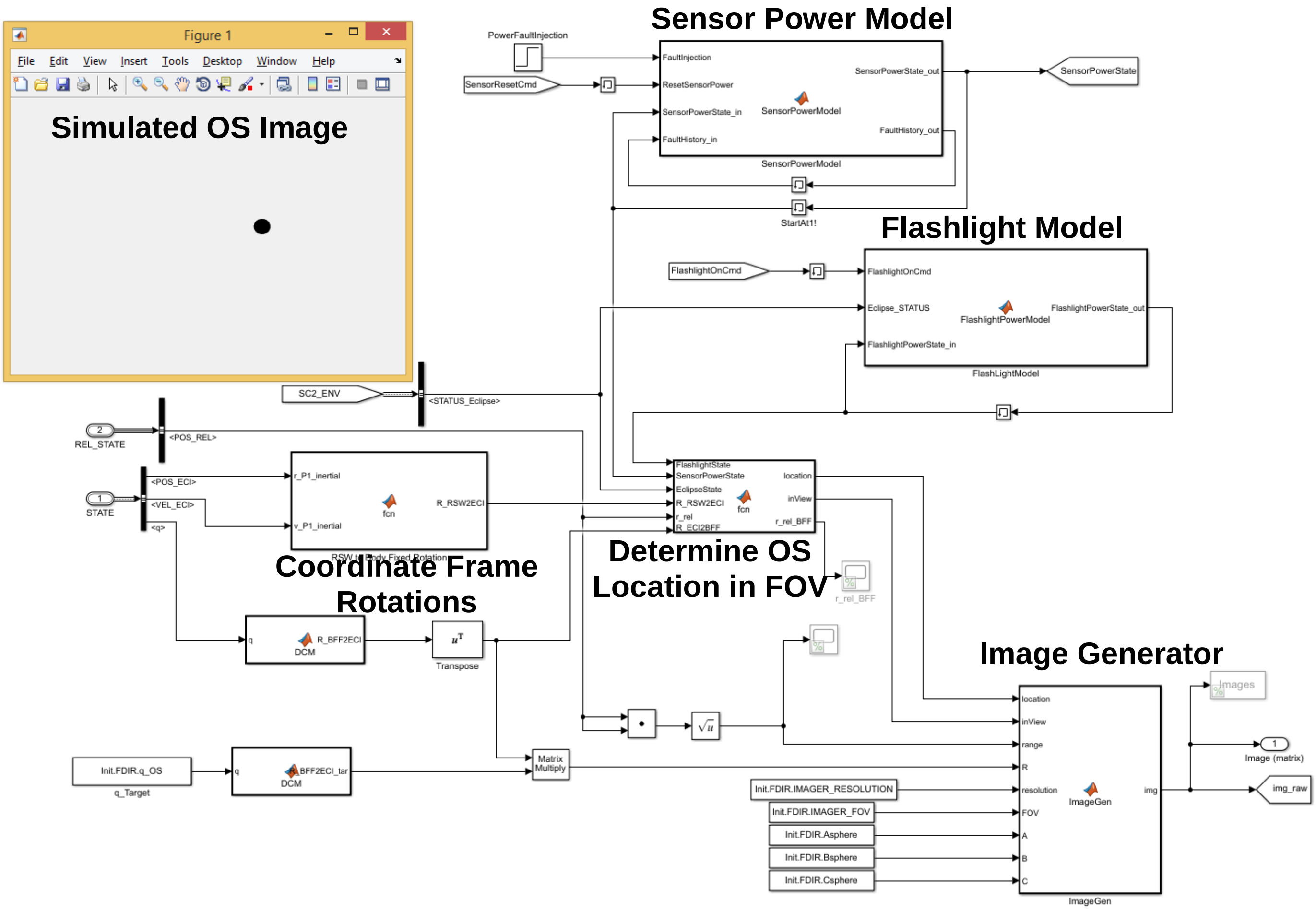


Figure 75 – Screenshot of rendezvous sensor model & simulated OS image

It should be noted that additional generic spacecraft sensor and actuator models are included in the SoftSim6D suite, but for simplicity in this study perfect state knowledge and control input execution is assumed (including impulsive ΔV maneuvers) and these sensor and actuator models are not utilized. Of course relaxing these assumptions would provide a greater degree of realism to the simulation, but for the faults examined in this study it was not necessary.

The next segment of the block diagram in Figure 74 is the Control System, which is arranged according to the generic state-based architecture containing components for state determination (evaluating evidence to determine the current state), state variables (maintaining state knowledge), and state control (computing control commands) [5]. There are two types of components in the Control System: GN&C Software Components which are focused on simulating the spacecraft's orbit & attitude control subsystem and Fault/Mode components which are used for FDIR and spacecraft mode management.

First, it should be noted that state determination for spacecraft dynamics and component states is *not* simulated explicitly through estimation algorithms. Instead, "truth" states from the simulation components are fed in directly to the control system as state variables. Dynamic states treated in this manner include spacecraft inertial and relative positions and velocities, attitudes and attitude rates, and eclipse/occultation status. This is in accordance with the assumption of perfect state knowledge mentioned above. Although navigation filter algorithms are not simulated, a variable representing filter convergence status is included in the simulation and is used to evaluate how the fault protection system responds to situations where such algorithms are not converged via user-defined fault injection. Sensor power status is also fed directly from the rendezvous sensor model into the Control System.

The spacecraft six-degree-of-freedom (6DOF) control includes two components: attitude target tracking and relative trajectory control. The attitude target tracking algorithm is identical to the algorithm developed for Prox-1 using a small satellite Control Moment Gyroscope [15], which points the camera boresight along the negative relative position vector. Because the simulation assumes perfect relative position and attitude knowledge, pointing the camera in this direction allows images of the OS to be generated by the rendezvous sensor model. As mentioned previously, perfect control input execution is assumed, so the torque commanded by the attitude target tracking algorithm is instantaneously imparted to the spacecraft attitude dynamics model. ROE-based relative trajectory control algorithms are also implemented to realize the concept of operations described in Section 4.4. Maneuvers are calculated dynamically and commanded onboard the SRO based on the current relative state rather than at preset times.

4.7.2 Simulation Results

A closed-loop V&V test of the FDIR system has been performed using SITL testing in the MATLAB/Simulink simulation described in the previous section. The initial relative orbit conditions are the same as those described in Section 4.4.3. The following rendezvous strategy parameters are also specified for the simulation. The maximum ΔV of the SRO is set to 0.25 m/s. This is used to determine when the SRO enters the Unavoidable Intercept Region. The capture distance is set to 1 m. This is used to determine when capture has occurred since the simulation does not include rigid body capture dynamics. The minimum safe distance is set to 10 m. This is used to determine the size of the safety ellipse for aborts. The standby distance is set to 50 m. This is used to determine when to end an abort; the SRO performs a maneuver to return to passively safe standby only after it has drifted this distance away from the OS in the along-track direction.

4.7.2.1 Nominal Approach

In Case 1, a nominal approach is performed, as shown in Figure 76. In this scenario, the SRO proceeds all the way to capture, with no faults injected and no recovery actions taken. A minimum range of 1 m occurs at 20,534 seconds, and the simulation ends to

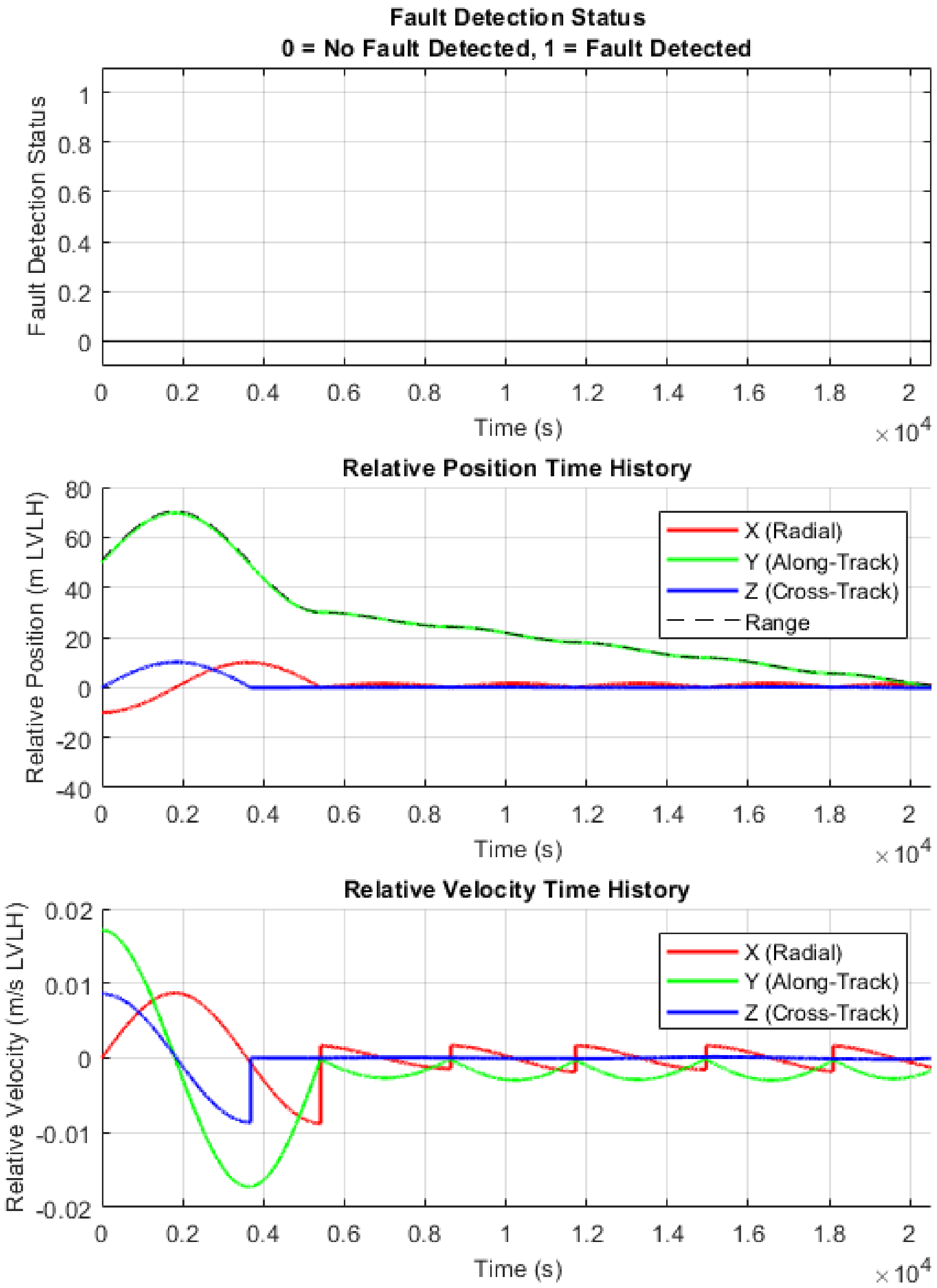


Figure 76 – Simulation Results for Case 1: Nominal Approach

represent capture. Note that this scenario is identical to the one presented in Section 4.4.4.1, except that no hold position maneuver is performed at the end of the simulation. However, unlike the previous scenario, this simulation does *not* assume linearized dynamics but propagates the full inertial orbits for both the SRO and the OS non-linearly and includes orbital perturbations. The simulation also includes a full 6DOF propagation including attitude dynamics.

4.7.2.2 Faults Prior to V-Bar Approach

Next, three scenarios are performed where different faults are injected during the planar hop prior to the start of the v-bar approach. In each of these scenarios the fault protection system successfully detects, diagnoses and responds to the fault and proceeds to capture the OS.

In Case 2, an angular rate fault results in the loss of the OS from the imager FOV. A fault is injected at 1,050 sec during the planar hop by turning off the attitude tracking controller. The OS slowly drifts out of the imager FOV until it is no longer visible, triggering a fault detection. The fault protection system then initiates a sky search slew, which scans the sky and quickly finds and tracks the OS again, as shown in the attitude and angular velocity plots in Figure 77. Note that only the portion of the simulation near the fault detection and recovery is shown for clarity. After reacquiring the OS in the imager FOV, the SRO continues to capture the OS at 20,534 sec and a minimum range of 1 m, as in Case 1. Note that the slew used for the sky search was designed for a small satellite with an agile attitude control system. For an actual SRO implementation, the slew rate would likely be much lower.

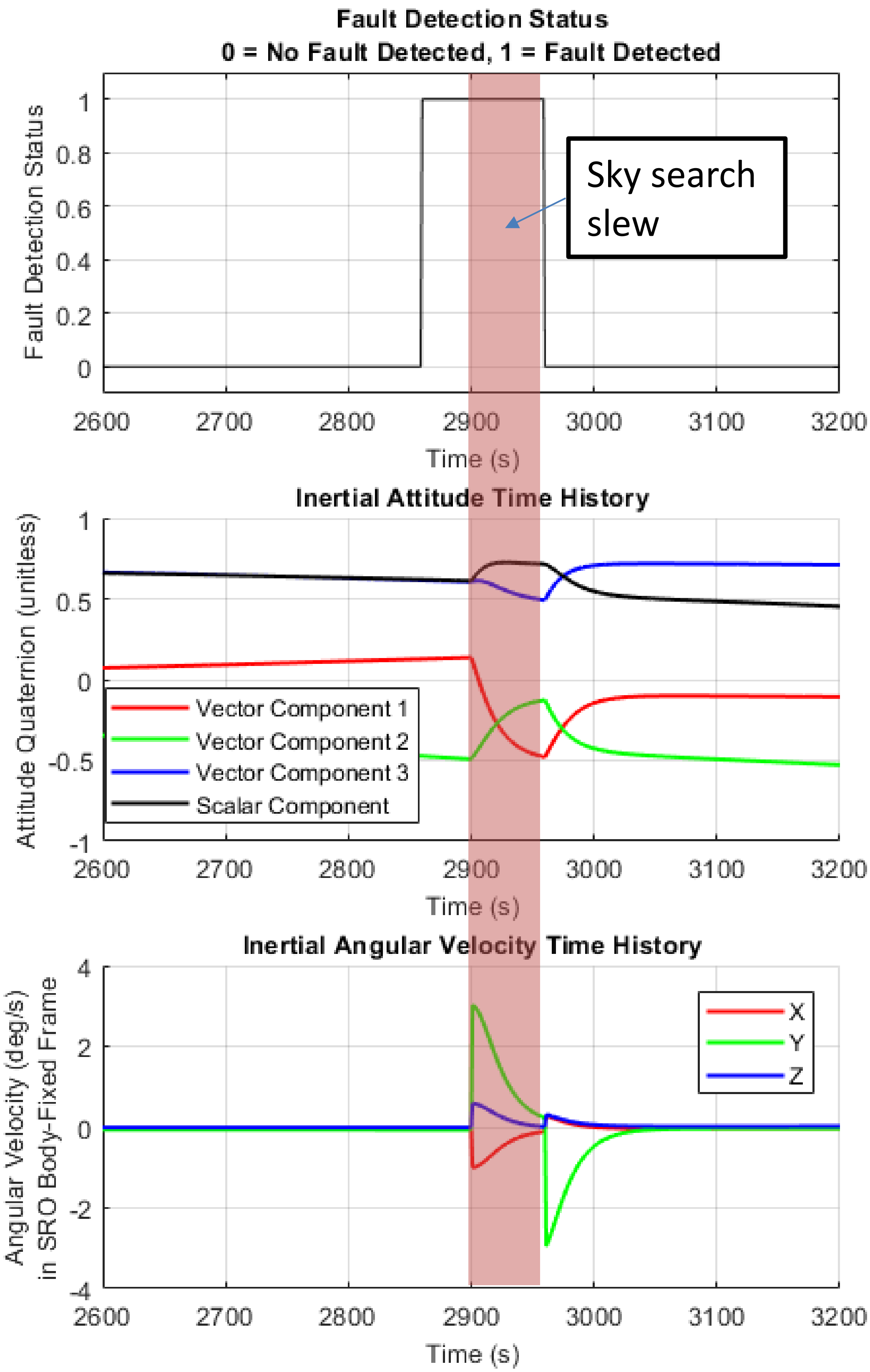


Figure 77 – Simulation Results for Case 2: Angular Rate Fault

In Case 3, the OS enters eclipse at 5,005 sec during the planar hop. The OS image is no longer visible in the sensor FOV, triggering a fault detection. Once the planar hop is complete, the system enters a position hold until eclipse ends before beginning the v-bar approach at 6,411 sec. This is seen in Figure 78 as a zero relative velocity in all axes and a constant relative position in all axes. After eclipse ends, the SRO continues to capture the OS at 22,978 sec and a minimum range of 1.4 m. Note that this scenario indicates that the SRO's autonomous system is "surprised" by the eclipse. Since eclipse is very predictable based on the Mars IAU ephemerides of the OS and the Sun, the autonomous system and ground support systems should be designed to anticipate and accommodate eclipse in the final rendezvous approach strategy.

In Case 4, a fault is injected at 5,400 sec during the planar hop indicating that the relative orbit determination filter is unconverged. The fault protection system detects this fault and commands a filter reset, which takes about 100 seconds to confirm. As in Case 3, the SRO enters a position hold until the fault is resolved before beginning the v-bar approach, as shown in Figure 79. Note that only the portion of the simulation near the fault detection and recovery is shown for clarity. The SRO then continues to capture the OS at 20,618 sec and a minimum range of 1 m.

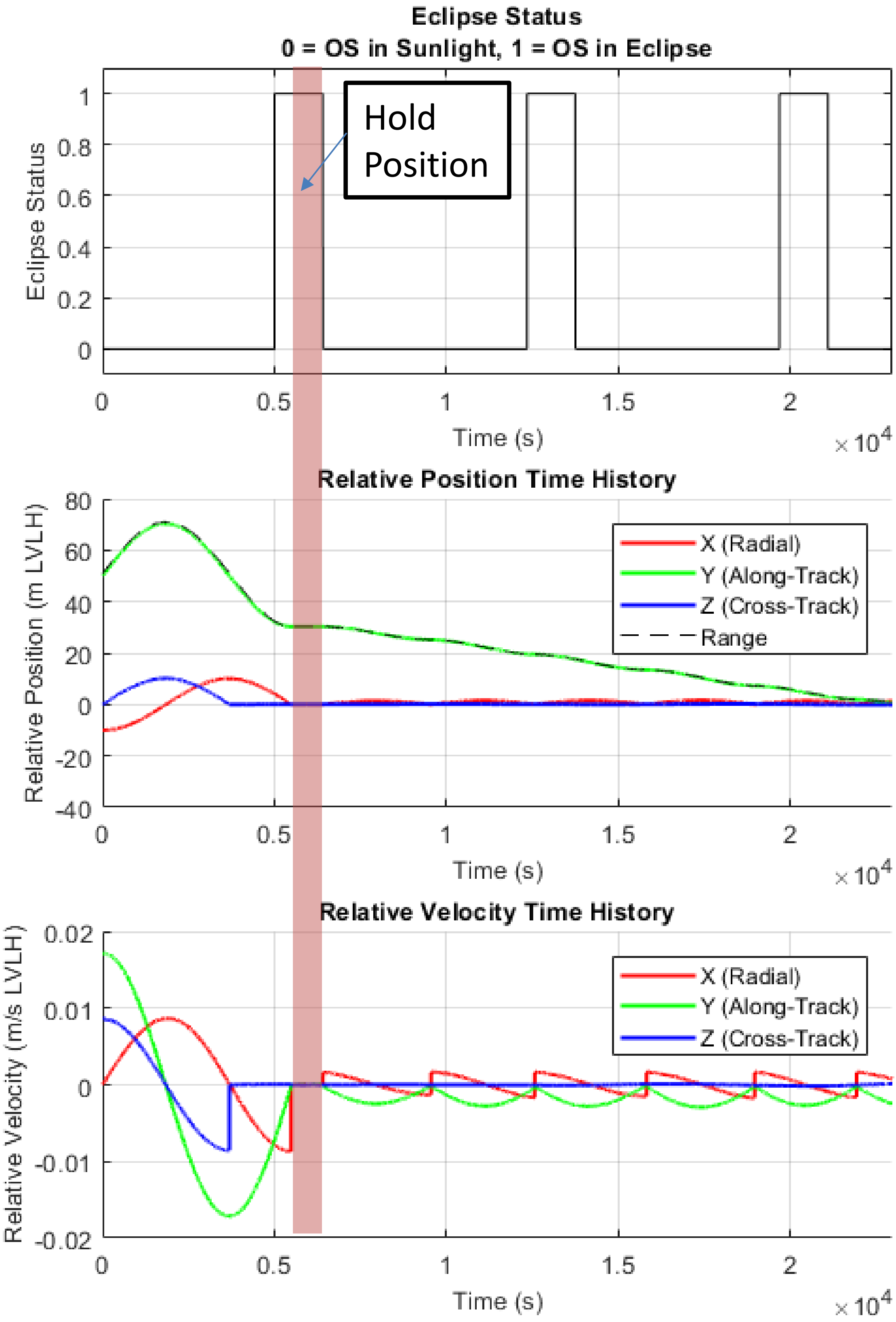


Figure 78 – Simulation Results for Case 3: OS Enters Eclipse

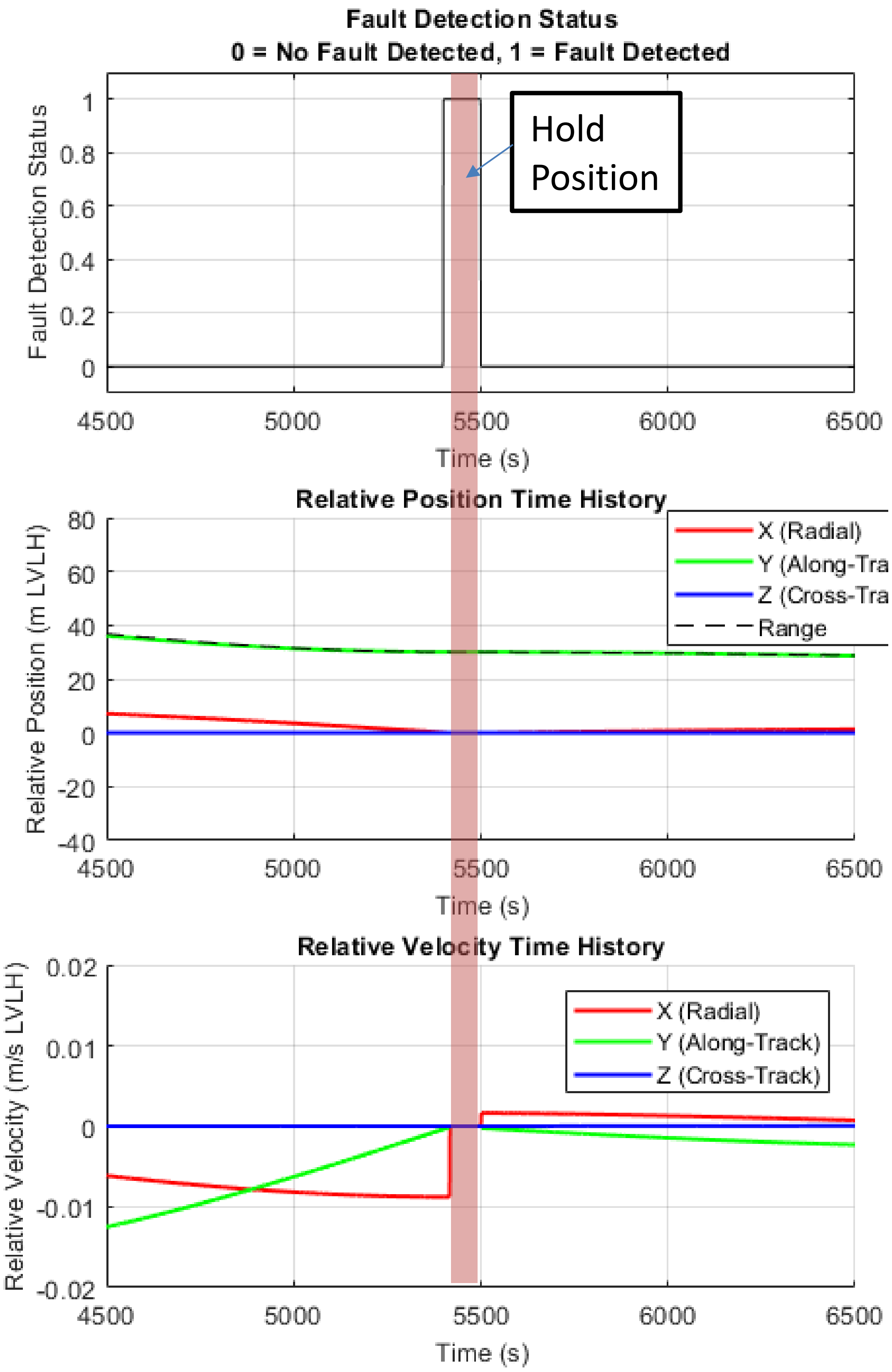


Figure 79 – Simulation Results for Case 4: Unconverged Relative Orbit Filter

4.7.2.3 Faults During V-Bar Approach

Finally, two scenarios are simulated where different faults are injected during the v-bar approach. In each of these scenarios, the fault protection system successfully detects the fault and immediately proceeds to a passive or active abort, bypassing the diagnostic step. This is the desired behavior specified by the requirements described in Section 4.5.

In Case 5, a fault is injected at 6,000 sec indicating that the relative orbit determination filter is unconverged, similar to Case 4. Unlike Case 4 however, in this scenario the planar hop has already been completed and the v-bar approach has begun *before* the fault is injected. The fault protection system detects the fault and immediately commands a passive abort because the SRO is in the Passive Miss Region (Zone 1) of the v-bar approach. The SRO then stops maneuvers and begins drifting. It passes through a minimum range of 24.23 m at 8,706 sec (about 30 minutes after the fault time). After this minimum range, the SRO drifts away from the OS in the negative along-track direction, as shown in Figure 80. As in the scenario described in Section 4.4.4.2, once the along-track distance reaches 50 m the SRO injects cross-track motion and returns to a passively safe out-of-plane standby trajectory.

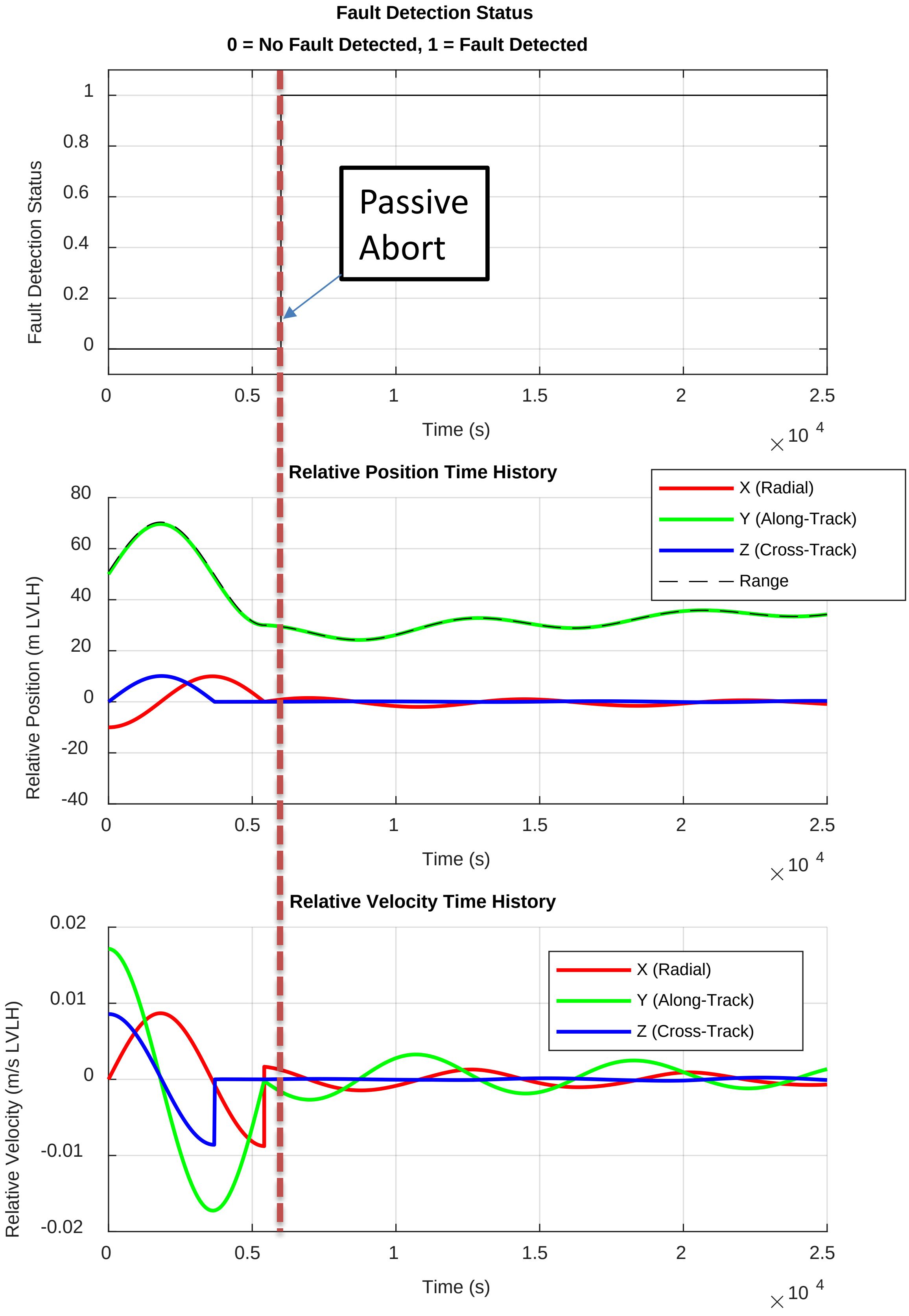


Figure 80 – Simulation Results for Case 5: Unconverged Relative Orbit Filter

In Case 6, a camera power fault is injected at 19,000 sec during the v-bar approach. The OS image is no longer visible in the sensor FOV, triggering a fault detection. The fault protection system detects this fault and immediately commands an active abort because the SRO is in the Active Abort Region (Zone 2) of the v-bar approach. The SRO then injects out-of-plane cross-track motion and begins drifting away from the OS in the negative along-track direction, as shown in Figure 81. This location becomes the point of closest approach at a range of 4.32 m. As in the scenario described in Section 4.4.4.3, once the along-track distance reaches 50 m the SRO freezes the along-track drift to return to the passively safe standby trajectory. A summary of all six simulation test cases is shown in Table 4.

Table 4 – Simulation test case summary

Case	**Description**	**Fault Time**	**Region Occurred**	**Response Taken**	**Minimum Range**	**Time of Min Range**
1	Nominal approach	n/a	n/a	n/a	1 m	20,534 sec
2	Angular rate results in loss of OS from FOV	1,050 sec	Planar Hop	Recovery	1 m	20,534 sec
3	OS enters eclipse	5,005 sec	Planar Hop	Recovery	1.4 m	22,978 sec
4	Unconverged Relative Orbit filter (before v-bar)	5,400 sec	Planar Hop	Recovery	1 m	20,618 sec
5	Unconverged Relative Orbit filter (during v-bar)	6,000 sec	Passive Miss Region	Passive Abort	24.23 m	8,706 sec
6	Camera power fault (during v-bar)	19,000 sec	Active Abort Region	Active Abort	4.32 m	19,000 sec

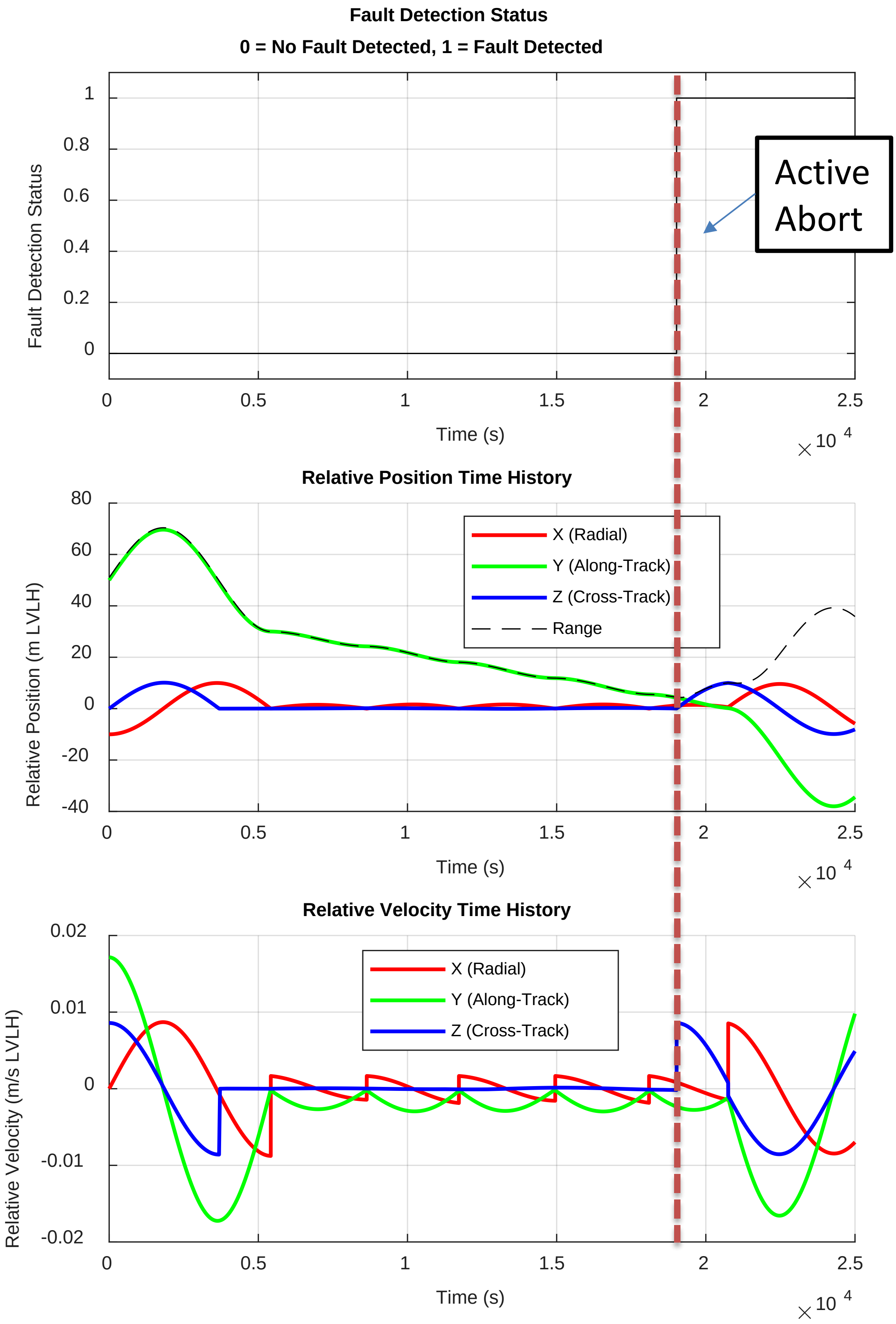


Figure 81 – Simulation Results for Case 6: Camera Power Fault

4.8 Conclusion

Each of the tasks described above has been completed successfully for an initial treatment of defining fault protection behavior for autonomous rendezvous and capture of the OS. A detailed fault tree has been defined, along with a detailed rendezvous and capture process concept of operations and a system block diagram. Initial fault protection strategies and requirements have been generated for a total of about 20 key faults from the various discipline areas. The architecture has been tested in simulation for several fault cases.

One goal of introducing fault protection earlier in the design cycle (during Pre-Phase A mission formulation) has been to help guide mission design considerations. One major observation is that the process of fault protection has been a forcing function for the MSR mission formulation team to clarify architecture and concept of operation decisions. In some cases, mission design and concept of operations assumptions have been documented for the first time. This has been an unexpected but welcome result, showing the value of fault protection not just as an add-on to a space mission design but as an essential component of the system design from the beginning.

Defining terminology clearly is very important. There have been miscommunications at several meetings because of different understandings for the definitions of certain terminology. For example, although terms like “guidance, navigation, & control” have fairly standard definitions, they may have different connotations in different contexts. Even the term “fault protection” means different things to different people. This challenge has been addressed by inviting open discussion and feedback in group meetings and by attempting to clarify any terms that could be confusing or misunderstood when they are presented. When developing tools like a fault tree, it is important to anticipate how terms will be understood by team members from various disciplines and define any terms that may be misinterpreted.

Another challenge has been determining what to do next when each step is completed. Since a new method of fault protection design is being experimented with, there is not a defined process to follow. A final challenge has been a backlog in the communication of progress throughout the project. Because of the cadence of meeting cycles, work was often completed several weeks before it could be communicated to all relevant stakeholders. These challenges have been addressed by seeking additional direction and advice of fault protection experts and rendezvous/capture subject matter experts. Their suggestions helped refine the direction of the study

CHAPTER 5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

This study has presented a fault protection architecture for aerospace vehicles that is generic, modular, and portable to flight software and enables model-based on-board fault diagnosis using the state machine paradigm. The architecture is generic and can be applied to any aerospace vehicle or mission. It features a generic simulation capability used for development, verification, & validation. Multiple applications have been used to demonstrate the generic utility of the architecture in simulation and flight tests. The architecture is also modular and contains components that can be added, removed, and rearranged easily. Environment models, vehicle sensor & actuator models, and dynamics models can be selected, modified, and rearranged in the simulation block diagram. Initial conditions, vehicle properties, and environmental scenarios can be easily redefined in an initialization script. The architecture is portable to flight software and it is straightforward to convert the initial design into flight software that is flown onboard the vehicle. An autocoding process has been defined and demonstrated for the Prox-1 small satellite.

The generic architecture is composed of five primary diagrams. A generic subsystem taxonomy defines the primary subsystems common to most aerospace vehicles and details common components for the guidance, navigation, & control subsystem. A generic fault tree analysis defines a process for determining which root cause and intermediate fault events could lead to an undesirable vehicle or mission failure. A generic functional state machine provides a model of vehicle mode state behavior by detailing processes common to many aerospace missions. A generic diagnostic state machine has been developed to enable on-board model-based diagnosis of faults. Lastly, a generic architecture block diagram illustrates how the fault and mode components work together with vehicle and environmental components to perform fault protection in simulation and in flight. A process has been defined for adapting the generic architecture to specific applications, and two case studies have been demonstrated for very different applications.

The first case study of the fault protection architecture is a terrestrial application for unmanned aerial vehicles known as the UAV Nervous System. The concept for the nervous system has been developed in collaboration with a company called FalconViz based at the King Abdullah University of Science and Technology in Thuwal, Saudi Arabia. A subsystem taxonomy and fault tree have been constructed and used to design a system to detect excess propeller vibration using supervised machine learning algorithms and alert the pilot by playing an audible signal from the radio hand controller. Upgrades have been added to the system to allow detection of propeller vibration faults, motor temperature faults, and electrical current/voltage faults on two arms of a six-arm rotary wing UAV simultaneously. These capabilities have been successfully demonstrated through flight testing. Finally, functional and diagnostic state machines have been created for the vibration detection case to develop a full fault protection architecture block diagram. The full architecture has been demonstrated in simulation using recorded flight test data. Flight tests and simulation using flight test data have demonstrated desired detection, diagnosis, and response performance for excess vibration faults and desired detection and response performance for temperature, current, and voltage faults.

The second case study of the fault protection architecture is a space-based proximity operations application for autonomous terminal rendezvous and capture of a

Mars Sample Return orbiting sample container. The concept and requirements for the fault protection architecture have been developed in collaboration with the Mars Sample Return team at the NASA Jet Propulsion Laboratory. A subsystem taxonomy, detailed rendezvous and capture fault tree, and concept of operations including a functional state machine have been developed with input from subject-matter experts at JPL. Detailed trajectory design has been completed for nominal approach, passive abort, and active abort scenarios, and autonomous trajectory control logic has been developed and demonstrated in simulation. A diagnostic state machine has been implemented in concert with diagnostic and response routines to detect and correct four distinct faults in various phases of autonomous approach. Each of these faults has similar detection criteria but distinct diagnostic and resolution processes. Abort logic has also been developed and demonstrated when faults occur during regions that risk the sample return orbiter colliding with the orbiting sample return canister. All of these components have been combined into an integrated fault protection architecture and demonstrated in simulation using realistic guidance, simulation, & control algorithms and components. Six simulation cases have been evaluated, and in each scenario the behavior of the fault protection architecture is consistent with desired results for fault detection, diagnosis, and recovery in accordance with the defined requirements.

In summary, a generic, modular, and portable architecture has been developed for aerospace vehicle fault protection. The architecture has been adapted to two distinct scenarios and has demonstrated the ability to successfully detect, diagnose, and respond to a variety of faults in real time using a state-based on-board system. Flight testing and detailed simulation have been used to thoroughly develop, verify, and validate this capability. A summary of all tasks for this investigation is shown in Table 5.

Table 5 – Task summary for this investigation

Status	Task	Completion Date
Complete	Initial FDIR literature review	Oct 2015
Complete	Initial FDIR architecture concept development	Apr 2016
Complete	UAV Nervous System proof-of-concept	July 2016
Complete	UAV Nervous System initial V&V flight tests	July 2016
Complete	UAV Nervous System conference paper [41]	Oct 2016
Complete	Dissertation proposal	Jan 2017
Complete	UAV Nervous System expansion	June 2017
Complete	UAV Nervous System final V&V flight tests	June 2017
Complete	MSR fault tree and requirements definition	July 2017
Complete	MSR FDIR conference paper [53]	Sept 2017
Complete	FDIR literature review revision	Dec 2017
Complete	MSR FDIR detailed design	Jan 2018
Complete	MSR FDIR SITL V&V	Mar 2018
Complete	Submit MSR FDIR peer-reviewed journal article [74]	Apr 2018
Complete	Generic FDIR architecture detailed design	Apr 2018
Complete	Demonstration of generic architecture with UAV flight test data	May 2018
Complete	Dissertation defense	June 2018
Complete	FDIR architecture peer-reviewed journal article	June 2018

5.2 Publications

In addition to the dissertation proposal and defense, several relevant publications are planned or completed. Other publications by the author are also listed.

Peer-Reviewed Journal Articles:

[15] P.Z. Schulte, D.A. Spencer, Development of an Integrated Spacecraft Guidance, Navigation, & Control Subsystem for Automated Proximity Operations, Acta Astronautica, 118 (Jan-Feb 2016), 168-186, doi:10.1016/j.actaastro.2015.10.010.

[31] D.A. Spencer, S.B. Chait, P.Z. Schulte, K.J. Okseniuk, M. Veto, Prox-1 University-Class Mission to Demonstrate Automated Proximity Operations, Journal of Spacecraft and Rockets, July 2016, doi:10.2514/1.A33526.

[74] P.Z. Schulte, D.A. Spencer, M. Goggin, Mars Sample Return Terminal Rendezvous Fault Protection, Journal of Spacecraft and Rockets, submitted Apr. 2018.

[75] P.Z. Schulte, D.A. Spencer, Generic State Machine Fault Protection Architecture for Aerospace Vehicle Guidance, Navigation, & Control, Journal of Aerospace Information Systems, submitted Jun. 2018.

Conference Papers Relevant to This Work:

[76] P.Z. Schulte, D.A. Spencer, Development of an Integrated Spacecraft Guidance, Navigation, & Control Subsystem for Automated Proximity Operations, 65th International Astronautical Congress, Toronto, Canada, Oct. 2014, IAC-14-C1.6.4x21108.

[77] K.J. Okseniuk, S.B. Chait, P.Z. Schulte, D.A. Spencer, Prox-1: Automated Proximity Operations on an ESPA Class Platform, 29th AIAA/USU Conference on Small Satellites, Logan, Utah, Aug. 2015.

[41] P.Z. Schulte, D.A. Spencer, N.G. Smith, M.F. McCabe, Development of a Fault Protection Architecture Based Upon State Machines, 67th International Astronautical Congress, Guadalajara, Mexico, Sept. 2016, IAC-16-D1.IP.2x32540.

[53] P.Z. Schulte, D.A. Spencer, State Machine Fault Protection for Automated Proximity Operations, 68th International Astronautical Congress, Adelaide, Australia, Sept. 2017, IAC-17-C1.5.11x36573.

[54] P.Z. Schulte, D.A. Spencer, Fault Protection for Mars Sample Return Autonomous Rendezvous & Capture, Symposium on Space Innovations, Atlanta, Georgia, Oct. 2017. (poster and presentation only)

[78] P.Z. Schulte, D.A. Spencer, On-Board Model-Based Fault Diagnosis for Autonomous Proximity Operations, 69th International Astronautical Congress, Bremen, Germany, Sept. 2018, IAC-18-C1.6x45016. (abstract accepted)

Other Conference Papers:

[79] P.Z. Schulte, J.W. Moore, A.L Morris, Verification and Validation of Requirements on the CEV Parachute Assembly System Using Design of Experiments, AIAA-2011-2558, 21st AIAA Aerodynamic Decelerator Systems Conference and Seminar, Dublin, Ireland, May 2011.

[80] P.Z. Schulte, E.G. Lightsey, K.B. Brumbaugh, R.L Staehle, Utilization of a Solar Sail to Perform a Lunar CubeSat Science Mission, 2nd Interplanetary CubeSat Workshop, Ithaca, New York, May 2013. (presentation only; paper withheld by sponsor)

[81] Pellegrino, M., Gibson, A., Mariscal, J.C., Schulte, P., “UNISPACE+50: Shared Vision, Common Action,” 68th International Astronautical Congress, Adelaide, Australia, Sept. 2017, IAC-17-E3.1.1x37185.

5.3 Suggestions for Future Work

This section presents possible extensions on this study. The suggestions for future work are organized based on the chapter divisions of this dissertation.

5.3.1 Generic Fault Protection Architecture

As described in Section 2.3.1, the subsystem taxonomy for the generic fault protection architecture lists generic subsystems for aerospace vehicles, but it only presents detailed generic components for the guidance, navigation, & control subsystem. It would be useful to add detailed components to the taxonomy for all subsystems. It addition, it would be beneficial to modify the generic subsystem taxonomy, fault tree, and functional state machine to be more object oriented and automated. It would also be helpful to define more detailed generic classes of faults for the fault tree and diagnostic state machine and link them to taxonomy and functional state machine. As indicated by Rasmussen, robust fault protection requires a measure of uncertainty [5], so it is desirable to formally define such a measure for use in fault diagnosis. Also, future researchers could utilize the generic fault protection architecture to implement and evaluate more advanced diagnostic capabilities [25], including constraint suspension [28-29] and other techniques demonstrated on Deep Space One [24] and Cassini [26,27]. The generic architecture could facilitate the use of these diagnosis methods for a wide variety of aerospace applications.

One possible method for these extensions would be to use the formal ontologies for fault protection and behavior modelling defined by JPL using SysML [12,22]. These formal ontologies might also be useful to define a more automated process to convert the generic architecture to a particular application and for developing content. For example, a future study could implement a "fault tree to state machine" algorithm to create the diagnostic state machine and link it to the functional state machine [40]. Such a process could also be used to populate each of the diagrams with content that is linked to other systems engineering processes for projects using formal methods for modelling. Finally, it would be immensely useful to extend the generic fault protection architecture beyond aerospace applications for domains such terrestrial robotics and autonomous cars.

5.3.2 UAV Nervous System

A next step to extend the UAV nervous system would be to expand the number of sensors with a full suite of vibration, temperature, current, and voltage sensors on each arm of the six arm hexacopter. Fault detection can be performed on all arms independently and simultaneously. Then, if a propeller imbalance or other fault is detected, the nervous system can indicate which propeller needs to be balanced. Similarly, temperature and current/voltage sensors can be placed on each motor and ESC and the nervous system can record all variables as a function of time and indicate to UAV designers and operators which components tend to overheat or break first.

Further sensors can also be added to expand the set of detectable failures. For example, many times when there is a problem with a UAV the first indication to the operators is an unusual sound. Thus, microphones collecting audio data near each propeller may be able to provide additional warning of faults. Data from microphones could be processed through a machine learning algorithm similarly to the acceleration data. Examples of other issues/failures on the UAVs that could be addressed by future work include monitoring the GPS receiver and magnetic compass (which can malfunction in flight, leading to the regularly reported problem of “fly-aways”), adding tachometers to detect changes in blade rotating speed, and ensuring healthy navigation filters (i.e. GPS position/velocity and attitude determination for roll/pitch/yaw angles and rates). Monitoring navigation variables would require communication with the flight controller, but redundant navigation systems could also aid in detecting navigation errors.

Another necessary update to the nervous system is smoothing out the startup process. Although quite convenient for prototyping and rapid development and testing, running Simulink in Windows onboard the copter is not a very elegant solution. It requires manually starting up Windows and initiating the Simulink model on a lab bench while connected to a monitor, then carrying the copter outside to begin flying. An intermediate step is to set up a high-definition video downlink to interact with Simulink in the field. Ultimately it would be desirable to remove the Windows/Simulink component from the system completely and perform all fault detection and data recording directly on the Teensy with a Secure Digital (SD) card shield attached. The Teensy can be fully customized by programming in C, and Simulink has the capability to generate C code via autocoding. If a simple KNN classification detection algorithm is implemented in C (or if an autocodable MATLAB algorithm is available), it can be integrated with autocode from Simulink and sensor interface code directly on the Teensy. This would streamline the process for using the nervous system and make it much easier to seamlessly integrate it with a copter.

5.3.3 Mars Sample Return Rendezvous & Capture

There is potential for some of the Mars Sample Return work to continue at JPL by adding more detail for the SRO rendezvous and capture FP strategy and by extending these FP concepts to other aspects of MSR concept development. Some of these methods could also be fed back into JPL's general fault protection processes to continue to advance state-based fault protection (especially diagnosis) for future missions. In retrospect, there are a few things that could be done differently in the Mars Sample Return study based on lessons learned. It would make sense to build the fault tree with a more functional structure rather than one based on rendezvous discipline areas. For example, if the OS is not seen in the rendezvous sensor's field of view, there could be an issue with relative orbit determination, attitude control, or sequencing that could cause this. The current version of the fault tree places this fault under relative orbit determination and not the other two branches. A more functional structure was suggested by a JPL fault protection expert, but the work was already far enough along that it was decided to leave the fault tree in its current format.

Additionally, several changes could be made to the simulation to make it more useful for Mars Sample Return mission designers. Adding maneuver execution error would make the sim more robust for evaluating additional faults. It would also be useful to include attitude and relative orbit filters for more realistic evaluation of fault protection performance with uncertain state information and for evaluating FDIR performance for filter faults. Collaborator McClain Goggin is developing a tool that allows the user to easily create and compare rendezvous trajectories and evaluate them based on passive safety and the probability of collision. In order to accurately evaluate the passive safety, the user will be able to select from a range of default sensor and filter models (or add their own) so that the state uncertainty covariance can be accurately determined for each case.

As mentioned in Section 4.7.2, several changes to the detailed implementation of the MSR fault protection architecture are necessary before a flight implementation is developed. The sky search slew rate used for reacquiring the OS when it drifts out of the sensor FOV should be adjusted for a large SRO spacecraft. Also, the autonomous system and ground support systems should be designed to anticipate and accommodate eclipse in the rendezvous approach strategy. In addition, detailed capture dynamics should be modeled and included in the analysis. Care should be taken when defining the transition between Zone 2 (Active Abort Region) and Zone 3 (Unavoidable Intercept Region) to ensure that active abort maneuvers have no chance of the SRO impacting the OS if the aborts are performed at a very close range. This is especially necessary if relative orbit information is lost during the abort. One way to account for this situation is to add margin to the capture distance, but regardless the amount of margin added should be determined by analysis.

APPENDIX A: AUTOCODING TECHNICAL MEMOS

This appendix is referenced by Section 2.2.3 and contains two technical memos written at Georgia Tech by the Prox-1 small satellite Guidance, Navigation, & Control team, of which the author was the subsystem lead. These memos describe the autocoding process which was developed for converting the GN&C subsystem code developed in MATLAB/Simulink to C/C++. Although MATLAB/Simulink has the capability to generate autocode, some small modifications and configuration setting changes were required for the process to work properly.

Technical Memorandum
December 2, 2014

TO: **Prox-1 Design Team**

FROM: **Meet Raj Patel & Jacob Sussman**

SUBJECT: **Autocoding of GN&C MasterSim**

REFERENCES:
(1) Fraticelli, J., "Simulink Code Generation: Tutorial for generating C code from Simulink Models using Simulink Coder," NASA Marshall Space Flight Center, 2012.
(2) "Call MATLAB Functions," Simulink Documentation, Mathworks, 2013. [http://www.mathworks.com/help/simulink/ug/calling-matlab-functions.html#bq1h2z9-48].
(3) "Simulink Documentation Center," Mathworks, 2013. [http://www.mathworks.com/help/simulink/index.html].

Purpose/Summary

This document provides guidelines for generating C code from an individual GN&C module and the entire developed GN&C Master Simulation of PROX-1. In order to successfully achieve the code generation process, specific steps have been documented along with appropriate reasoning. Common errors are also documented in this report. The report is divided in two main sections:

1. Assembling & Running the simulation
2. Autocoding procedure

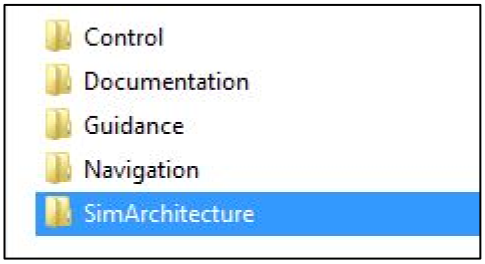
- ➢ For this report, examples are given specifically using the "HIL_6DOF_MasterSim3-_APF_NoNav" Simulink model. Autocoding any other MasterSim should be fairly similar to this report's design. Considerations for future MasterSim releases are addressed when appropriate and are preceded with the same bullet point used for this section.

[NOTE: Good ideas that have yet to be implemented or properly looked into will be encased in square brackets and preceded by "NOTE:" in bold letters. If agreeable to those in command, they should be implemented in the Spring 2015 semester.]

Assembling & Running the Simulation

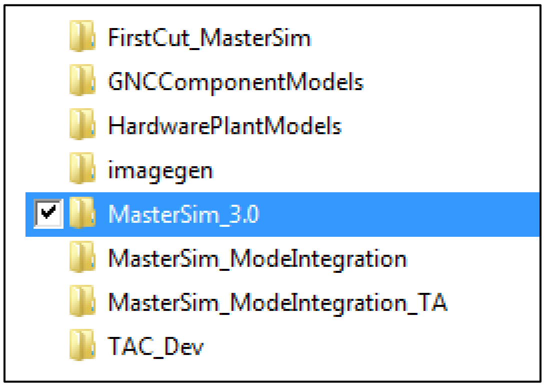
Steps:

- Open T-square, and download GNC_Current folder. Make sure to download the recent working folder of the simulation. After downloading and unzipping the file check that it has the following subfolders: Control, Documentation, Guidance, Navigation and SimArchitecture. Re-download if necessary. Open the "SimArchitecture" folder. The image shown below shows the subfolders.



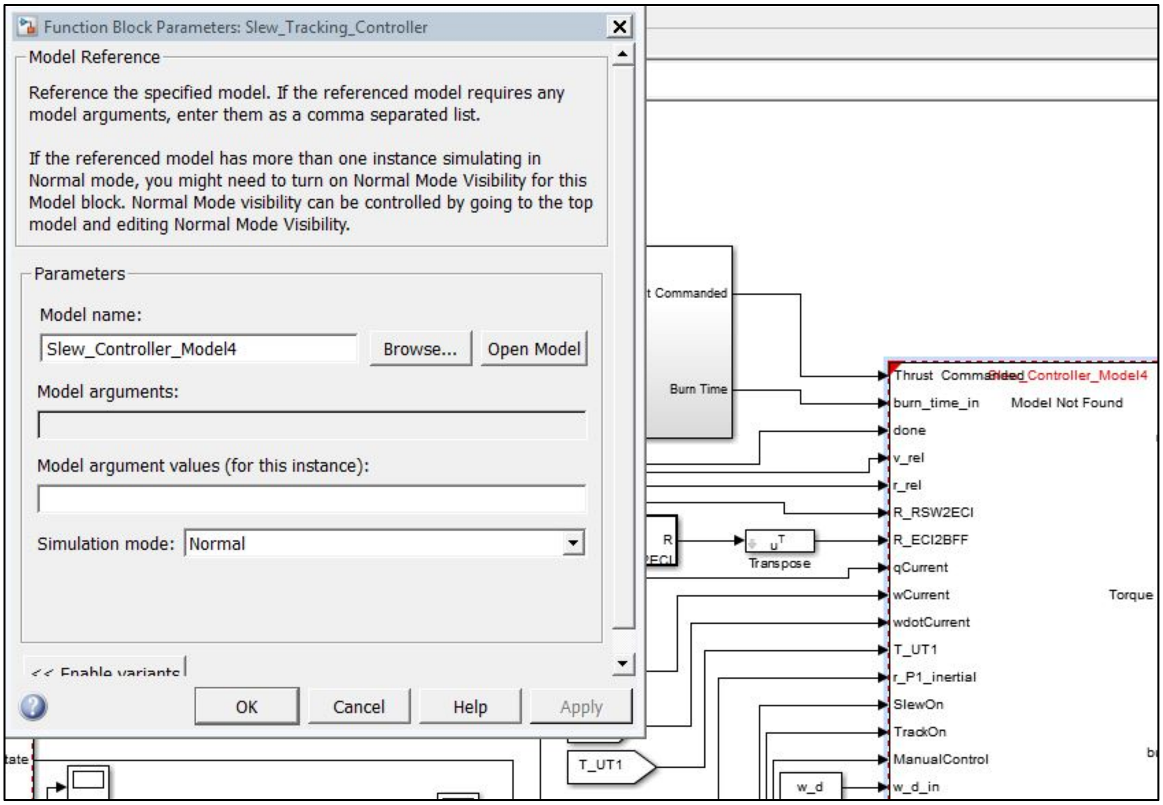
Control
Documentation
Guidance
Navigation
SimArchitecture

- Access "MasterSim_3.0" folder and open "HIL_6DOF_MasterSim3_APF_NoNav" Simulink model. Multiple graphic windows, Matlab main window and Simulink main model should be available.



FirstCut_MasterSim
GNCComponentModels
HardwarePlantModels
imagegen
☑ MasterSim_3.0
MasterSim_ModelIntegration
MasterSim_ModelIntegration_TA
TAC_Dev

- In the "HIL_6DOF_MasterSim3_APF_NoNav" Simulink window there should be few un-referenced blocks. There modules appear to look red since model-referencing is required. Right click on those blocks (the one with red dotted blocks), and select "Block Parameter (model referencing)". The window may appear as shown below:



Function Block Parameters: Slew_Tracking_Controller

Model Reference

Reference the specified model. If the referenced model requires any model arguments, enter them as a comma separated list.

If the referenced model has more than one instance simulating in Normal mode, you might need to turn on Normal Mode Visibility for this Model block. Normal Mode visibility can be controlled by going to the top model and editing Normal Mode Visibility.

Parameters

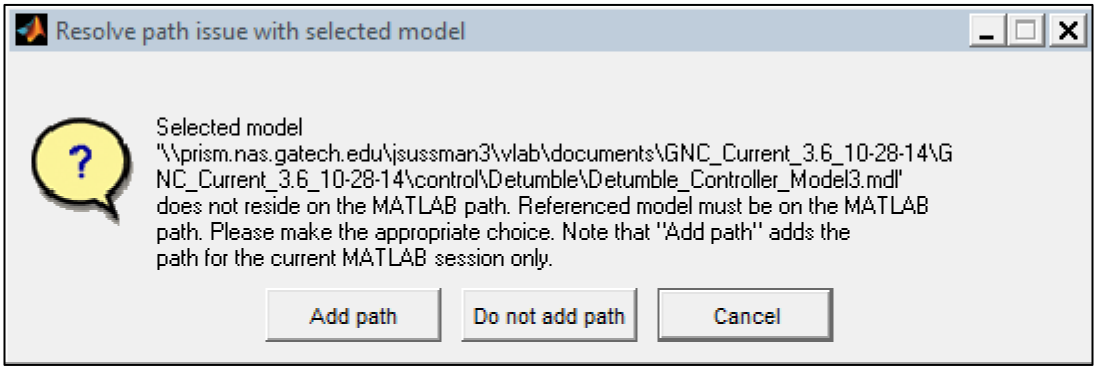
Model name: Slew_Controller_Model4

Model arguments:

Model argument values (for this instance):

Simulation mode: Normal

- The preceding image shows the model reference block which includes “Model name” under “Parameters” window. Click on “Browse” and select the file name corresponding to the model name. For instance, “Slew_Controller_Model4” file Simulink model file is selected from the main GNC Control folder. After the file is selected, a prompt may ask to add the selected model to the current path. Click “Add path” and continue. The red dotted module will now turn its boundary into a darker one to signify that the model is found and ready to be used.

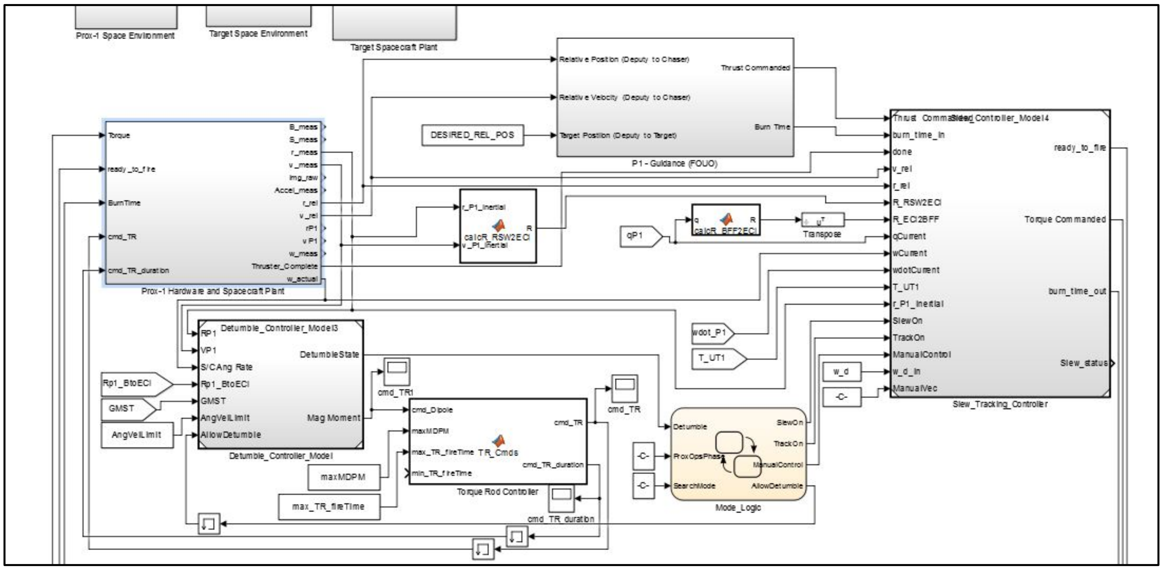


Resolve path issue with selected model

Selected model '\\prism.nas.gatech.edu\jsussman3\vlab\documents\GNC_Current_3.6_10-28-14\GNC_Current_3.6_10-28-14\control\Detumble\Detumble_Controller_Model3.mdl' does not reside on the MATLAB path. Referenced model must be on the MATLAB path. Please make the appropriate choice. Note that "Add path" adds the path for the current MATLAB session only.

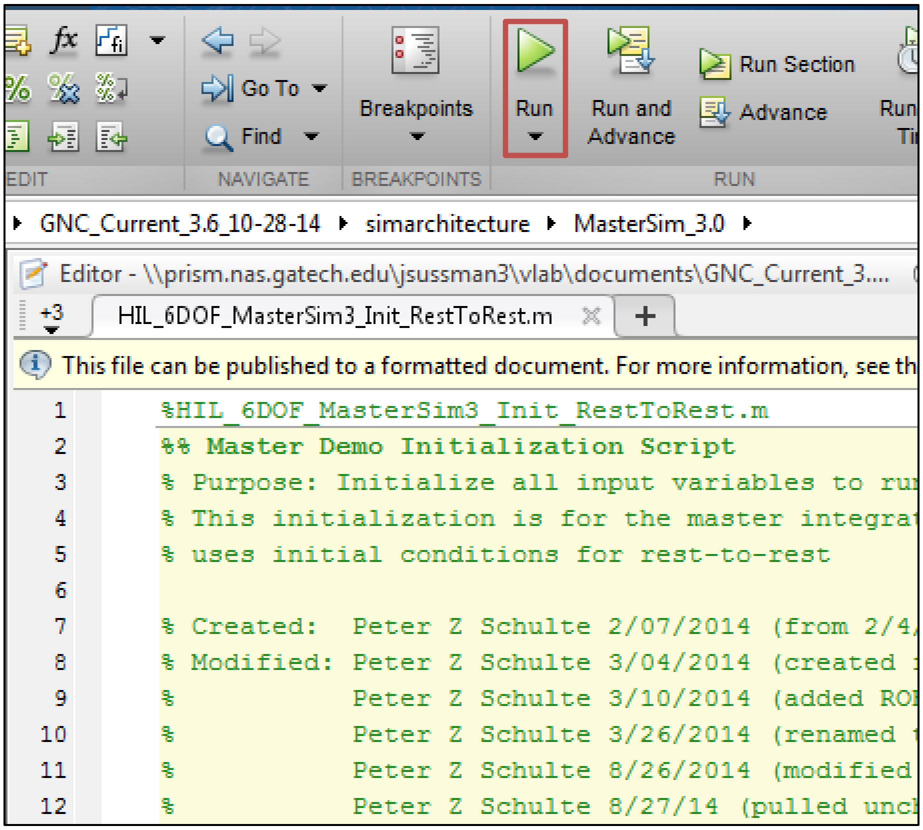
- Follow the similar procedure for rest of the modules with red dotted unreferenced model. For current Sim model, model referencing should be made for “Slew&Tracking” model, “Detumble” model and Prox-1 Hardware and spacecraft plant model. In order to reference the Prox-1 Hardware and spacecraft plant model, click on the model. This will open a sub-model consisting of “Prox-1 Hardware Models”, “Prox-1 Spacecraft Plant”, “Prox-1 Sensor Suit” and “Power Production Models”. Select the “Prox-1 Hardware Models”.

- Inside this “Prox-1 Hardware Models” there are 4 sub-models. ‘Thurster_SMC’,’TorqueRod_SMC’,’UT_Austin_Thruster’ and ‘TorqueRod_Hardware’. Referencing each model is not required; referencing just one model out of 4 will suffice. This is because each of these models are in the same folder, so that when the path to the folder is added for one of the models Simulink automatically detects the others. The Final Sim is shown below:



  - ➢ Future MasterSim releases will have more externally-referenced sub-models, but the process for adding them is the same: click the broken model link, find the model, add the folder path.

- Time to initialize the workspace environment. Make sure to run the initialization file from the current folder in main Matlab window. For this demonstration, open any of the following matlab.m files located in the same folder as the MasterSim model: HIL_6DOF_MasterSim3_Init_NMC.m, HIL_6DOF_MasterSim3_RestToRest.m, or HIL_6DOF_MasterSim3_Init_LeadingOrbit.m. Running one of these files will initialize various model simulation parameters, e.g., the mass of the spacecraft, the target’s location, or a simulated initial noise matrix.



GNC_Current_3.6_10-28-14 ▸ simarchitecture ▸ MasterSim_3.0 ▸

Editor - \\prism.nas.gatech.edu\jsussman3\vlab\documents\GNC_Current_3....

HIL_6DOF_MasterSim3_Init_RestToRest.m

This file can be published to a formatted document. For more information, see th

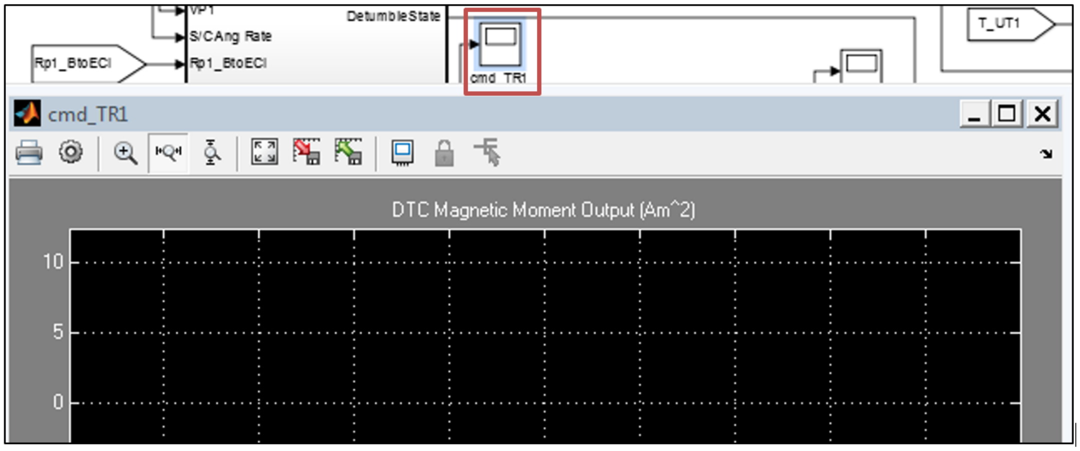
```
%HIL_6DOF_MasterSim3_Init_RestToRest.m
%% Master Demo Initialization Script
% Purpose: Initialize all input variables to ru
% This initialization is for the master integrat
% uses initial conditions for rest-to-rest

% Created:  Peter Z Schulte 2/07/2014 (from 2/4,
% Modified: Peter Z Schulte 3/04/2014 (created
%           Peter Z Schulte 3/10/2014 (added RO
%           Peter Z Schulte 3/26/2014 (renamed
%           Peter Z Schulte 8/26/2014 (modified
%           Peter Z Schulte 8/27/14 (pulled unc
```

- ➢ Later MasterSim versions may require more than one initialization file to run. For example, the current version of MasterSim 4 (as of December 2014) requires an additional “SpacecraftPlant_EnvironmentModel_Init.m” file to be run pre-simulation in order to function properly. If you are unsure about how many initialization files are required for the current MasterSim ask someone in the know. [NOTE: An up-to-date document should be put in the MasterSim folder detailing the initialization files needed to properly run the current MasterSim.]

- Go back to the main Simulink file and run the simulation. Any errors cause during the simulation will be related to errors in the .m input file or a built error of Simulink. Any other source of error in running the simulation should be addressed and tackled by interpreting the error details provided by Matlab.

- If the run is successful then the plots of simulation variables will update with each time-step. If you can, check that the plots make sense. If you already closed every plot you can reopen a target plot by double-clicking on the respective scope block, as shown below:



Autocoding Procedure

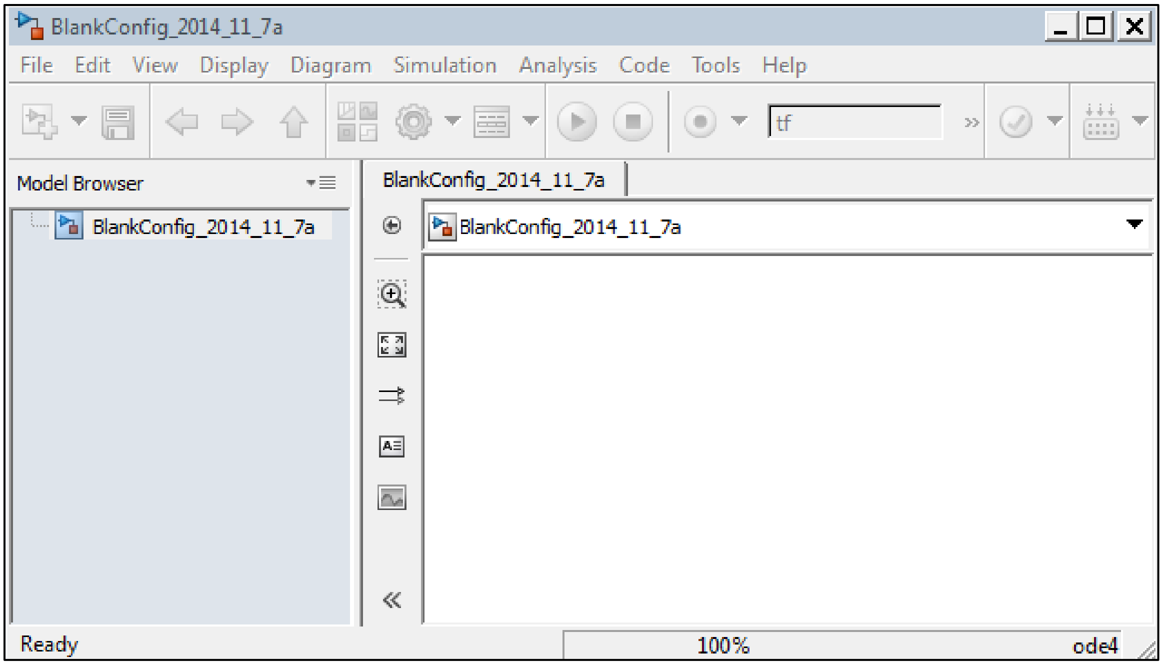
Now that the simulation is up and running the autocoding process can begin. The principles of this process can be distilled into 5 parts:

1. *Documentation*: Organizing what you learned from trial and error. Prevents re-inventing the wheel. See: this report.
2. *Configuration*: Certain configuration parameters should be in place for a smooth autocoding experience. Changing these parameters (especially the hardware configuration settings) **will** initially prevent proper simulation of the model.
3. *Simulation*: After setting the configuration parameters, the model needs to be set up again so that it can properly run as before. This time, however, the model can go through the Code Generation process.
4. *Generation*: Code Generation is the cornerstone of autocoding. If the model is set up right, then the Simulink Coder should be able to convert the simulation into C code.
5. *Compilation*: The final autocoding step consists of compiling the C code to run on the BeagleBoard and adding possibly preprocessor directives to analyze the C model in real-time.

Configuration

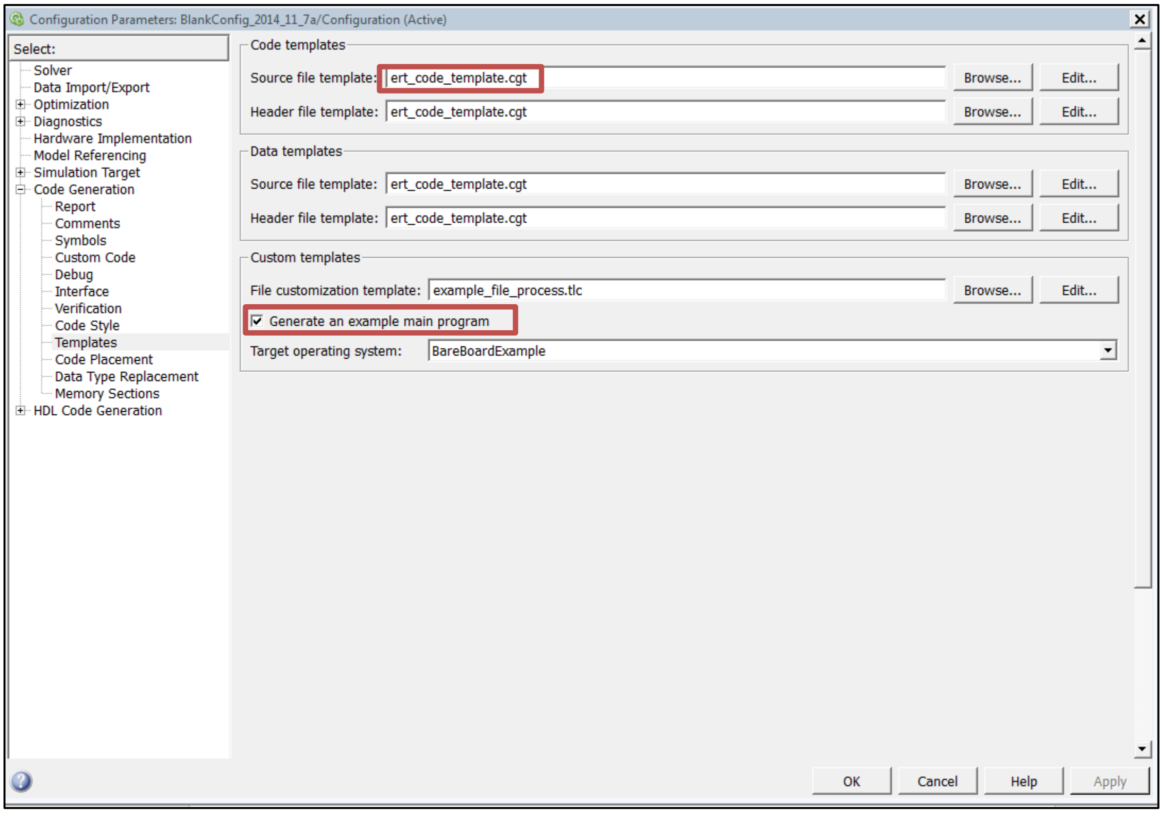
Steps:

- Locate the BlankConfig file. This is a Simulink model that is entirely empty save for its configuration parameter settings. This BlankConfig file was created in order to speed up the Configuration part of the Autocoding process. Simply copy the entire MasterSim and paste it into the BlankConfig file. Close the original model and save the newly-pasted model as the original MasterSim file (or some closely named variant thereof).



- If this is done correctly and the BlankConfig file has the right settings then Configuration is done. When dealing with new models, however, the settings may not be properly set up. What follows is a guide for what configuration settings are some of the most important.

- To access a model’s Configuration Settings first click the ‘Simulation’ tab and then click the ‘Model Configuration Parameters’ button. A GUI should open containing the configuration settings. With ‘Solver’ selected the source file template is chosen as ‘ert_code_template.cgt’. ‘.cgt’ stands for Code Generation Template, and ‘ert’ stands for Embedded Real Time. This file contains a template for the Coder to base its code generation off of. This .cgt file must be selected for other settings to function (notably the package and zip function which only appears to exist in this template file). ‘Generate an example main program’ is checked in order to simplify joining the C files together.


Configuration Parameters: BlankConfig_2014_11_7a/Configuration (Active)

Code templates

Source file template: ert_code_template.cgt

Header file template: ert_code_template.cgt

Data templates

Source file template: ert_code_template.cgt

Header file template: ert_code_template.cgt

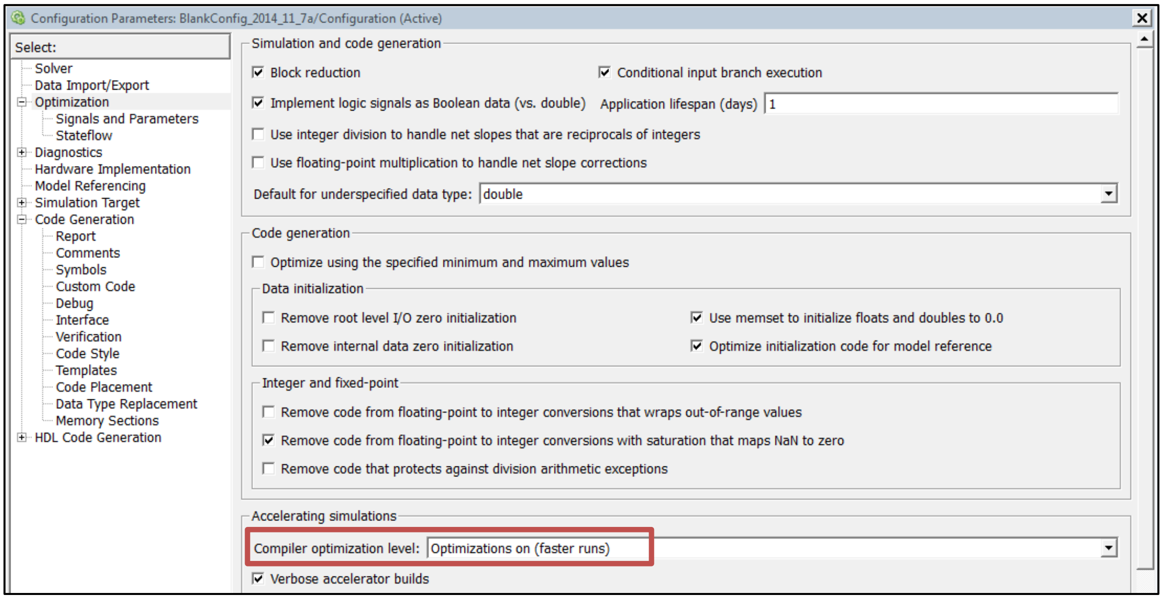
Custom templates

File customization template: example_file_process.tlc

☑ Generate an example main program

Target operating system: BareBoardExample

- In the ‘Optimization’ selection, and under ‘Accelerating simulations’ the Compiler optimization level is set for faster runs. This will make the building of the C code take longer but the resulting code will be more efficient. Taking more time now in order to save time once the mission is in effect.


Configuration Parameters: BlankConfig_2014_11_7a/Configuration (Active)

Simulation and code generation

☑ Block reduction

☑ Conditional input branch execution

☑ Implement logic signals as Boolean data (vs. double)

Application lifespan (days) 1

☐ Use integer division to handle net slopes that are reciprocals of integers

☐ Use floating-point multiplication to handle net slope corrections

Default for underspecified data type: double

Code generation

☐ Optimize using the specified minimum and maximum values

Data initialization

☐ Remove root level I/O zero initialization

☑ Use memset to initialize floats and doubles to 0.0

☐ Remove internal data zero initialization

☑ Optimize initialization code for model reference

Integer and fixed-point

☐ Remove code from floating-point to integer conversions that wraps out-of-range values

☑ Remove code from floating-point to integer conversions with saturation that maps NaN to zero

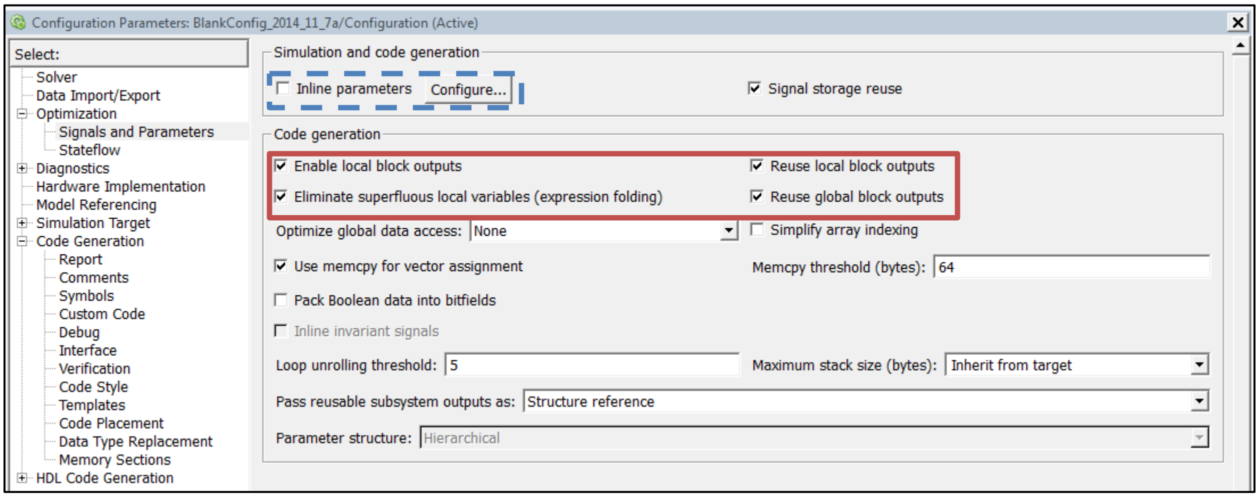
☐ Remove code that protects against division arithmetic exceptions

Accelerating simulations

Compiler optimization level: Optimizations on (faster runs)

☑ Verbose accelerator builds

- In the ‘Signals and Parameters’ menu, ‘Enable local block outputs’, ‘Reuse local block outputs’, ‘Eliminate superfluous local variables (expression folding)’, and ‘Reuse global block outputs’ are all checked in order to make the C code more efficient. [NOTE: Flight Software has expressed an interest in the use of Inline parameters. Effective use of inline parameters can results in cleaner, faster C code. Configuring inline parameters should be looked into.]



Configuration Parameters: BlankConfig_2014_11_7a/Configuration (Active)

Simulation and code generation

□ Inline parameters Configure...

☑ Signal storage reuse

Code generation

☑ Enable local block outputs

☑ Reuse local block outputs

☑ Eliminate superfluous local variables (expression folding)

☑ Reuse global block outputs

Optimize global data access: None

□ Simplify array indexing

☑ Use memcpy for vector assignment

Memcpy threshold (bytes): 64

□ Pack Boolean data into bitfields

□ Inline invariant signals

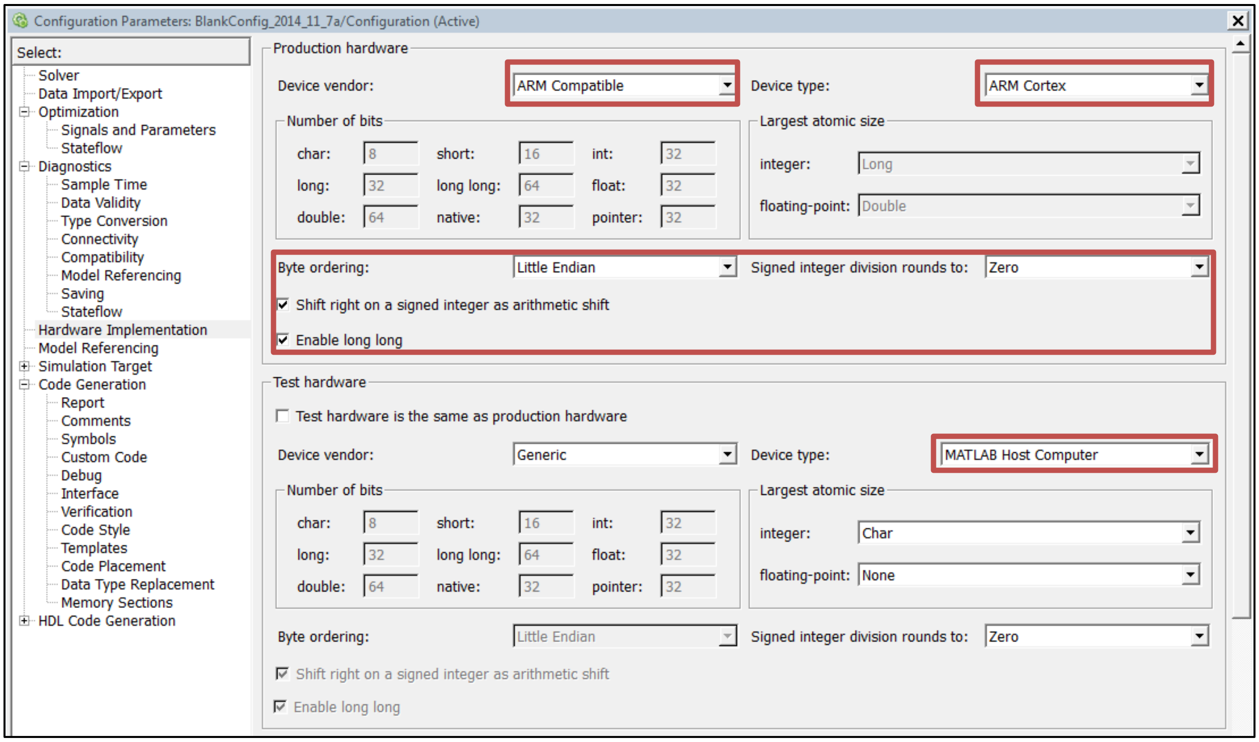
Loop unrolling threshold: 5

Maximum stack size (bytes): Inherit from target

Pass reusable subsystem outputs as: Structure reference

Parameter structure: Hierarchical

- Under ‘Hardware Implementation’ the right production hardware needs to be selected for the BeagleBoard XM. It has an ARM Cortex processor and each option for this device has been personally checked with the hardware itself to make sure they are correct for the BeagleBoard. The byte ordering is Little Endian. The signed integer division rounds to zero. The signed integer arithmetic shift is a shift right. And long long is an enabled variable. The test hardware should be set to the MATLAB Host Computer. The ‘Hardware Implementation’ menu is what messes up the ability for the model to simulate as before. In the simulation section it will be explained how to overcome this issue.



Production hardware

Device vendor: ARM Compatible Device type: ARM Cortex

Number of bits

char: 8 short: 16 int: 32
long: 32 long long: 64 float: 32
double: 64 native: 32 pointer: 32

Largest atomic size

integer: Long
floating-point: Double

Byte ordering: Little Endian Signed integer division rounds to: Zero

☑ Shift right on a signed integer as arithmetic shift

☑ Enable long long

Test hardware

☐ Test hardware is the same as production hardware

Device vendor: Generic Device type: MATLAB Host Computer

Number of bits

char: 8 short: 16 int: 32
long: 32 long long: 64 float: 32
double: 64 native: 32 pointer: 32

Largest atomic size

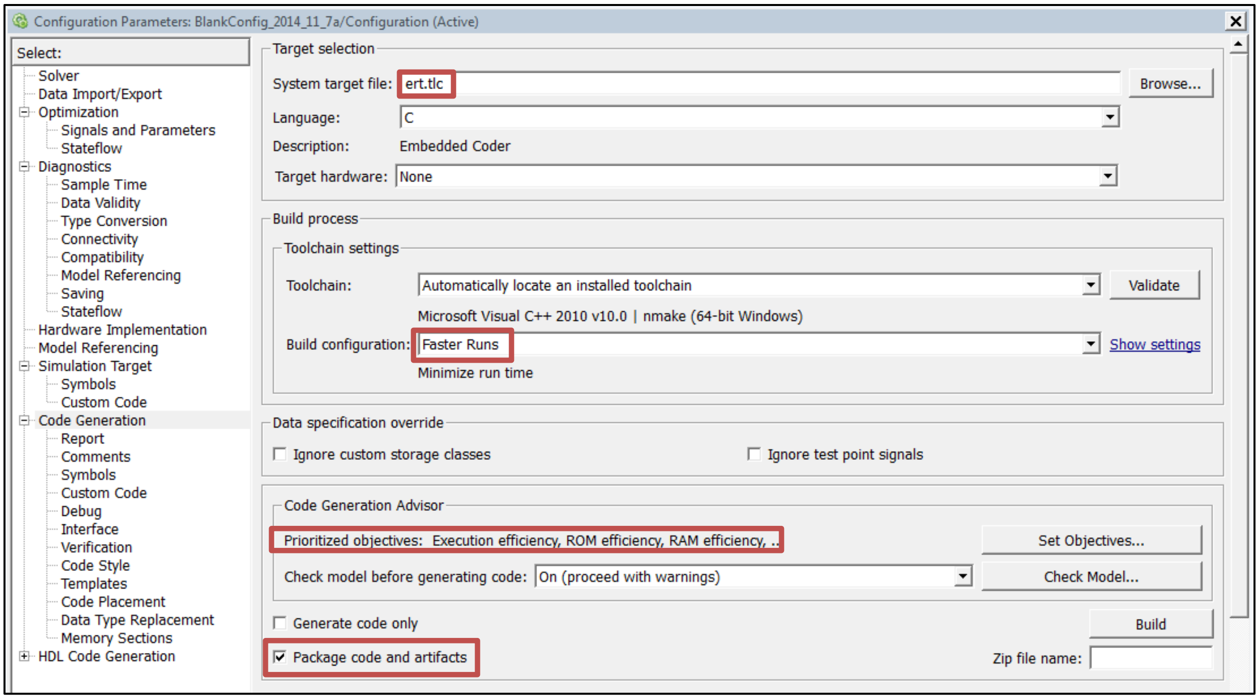
integer: Char
floating-point: None

Byte ordering: Little Endian Signed integer division rounds to: Zero

☑ Shift right on a signed integer as arithmetic shift

☑ Enable long long

- Similar to previous options, under ‘Code Generation’ the system target file must be set to ‘ert.tlc’ in order for certain settings to be active, ‘Build Configuration’ should be set to ‘Faster Runs’ to trade off time wasted now versus time wasted later, and the ‘Prioritized Objectives’ are set to Execution, RAM and ROM efficiency to speed up the C code. Also in this menu, the ‘Package code and artifacts’ **must** be checked. It forces the Coder to zip up every required file for the C program into one zip file. If this is not checked then you must individually find certain header files in your computer’s MATLAB root. This is not an easy task, and it is arduous work.


Configuration Parameters: BlankConfig_2014_11_7a/Configuration (Active)

Target selection

System target file: ert.tlc

Language: C

Description: Embedded Coder

Target hardware: None

Build process

Toolchain settings

Toolchain: Automatically locate an installed toolchain

Microsoft Visual C++ 2010 v10.0 | nmake (64-bit Windows)

Build configuration: Faster Runs

Minimize run time

Data specification override

□ Ignore custom storage classes

□ Ignore test point signals

Code Generation Advisor

Prioritized objectives: Execution efficiency, ROM efficiency, RAM efficiency, ..

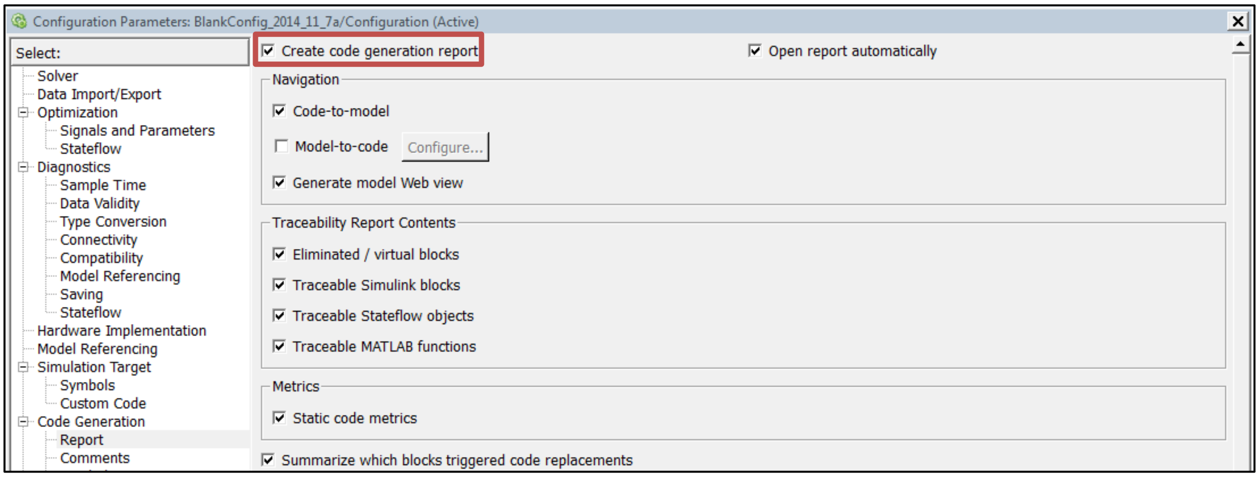
Check model before generating code: On (proceed with warnings)

□ Generate code only

☑ Package code and artifacts

Zip file name:

- Finally, another very helpful option to check off is ‘Create code generation report’ located under ‘Code Generation’ and ‘Report’. This allows you to see the generated C code in a user-friendly way once the code is finished building.


Configuration Parameters: BlankConfig_2014_11_7a/Configuration (Active)

☑ Create code generation report

☑ Open report automatically

Navigation

☑ Code-to-model

□ Model-to-code

☑ Generate model Web view

Traceability Report Contents

☑ Eliminated / virtual blocks

☑ Traceable Simulink blocks

☑ Traceable Stateflow objects

☑ Traceable MATLAB functions

Metrics

☑ Static code metrics

☑ Summarize which blocks triggered code replacements

- ➢ Future versions of MasterSim will likely break with these current settings. Each new model requires understanding and configuring more new configuration parameters. This is one of the most tedious parts of the autocoding process (Aside from waiting for the actual code to generate).

Simulation

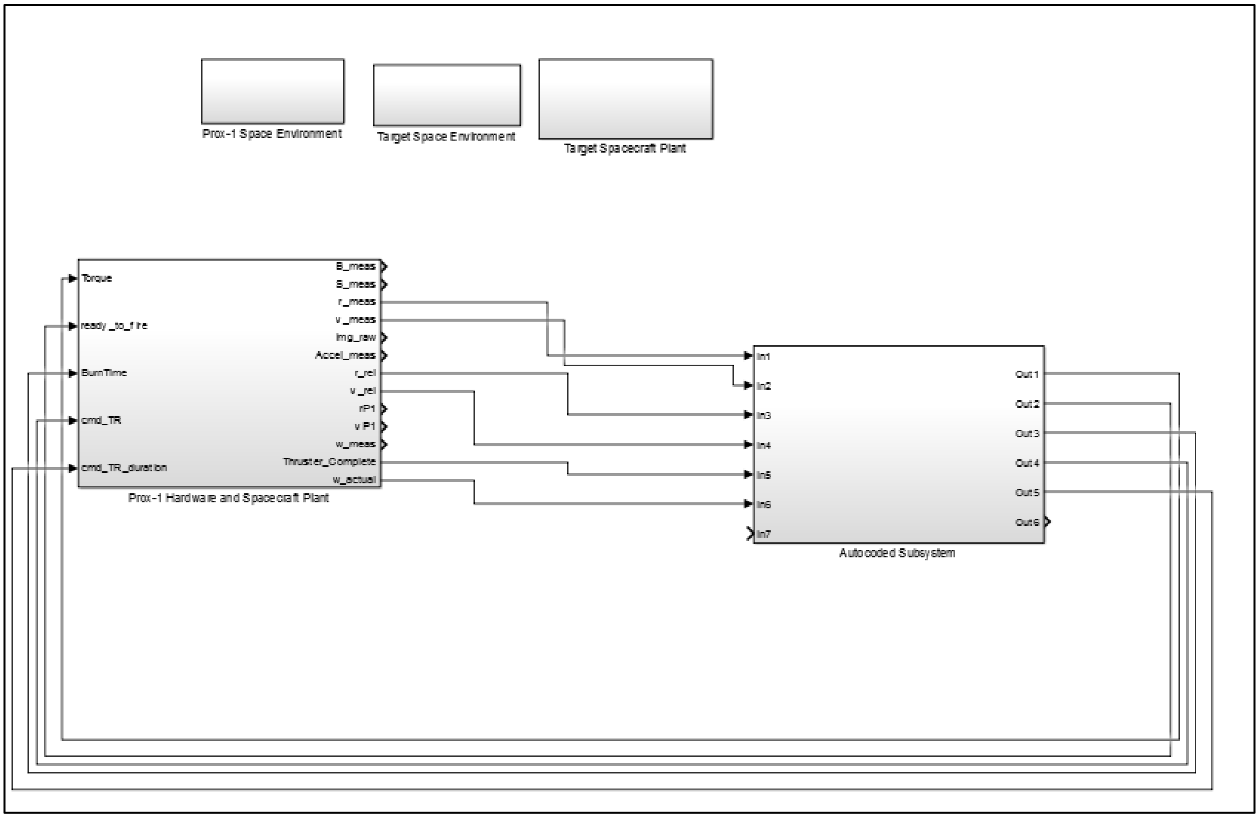
Steps:

- With the new Configuration settings the MasterSim will likely not run. This is because changing the settings for the MasterSim does not automatically change the settings in all of the externally-linked submodels. Clashing settings crashes the simulation. And if you can't run the simulation then you definitely can't generate code from it. To fix this, open up an offending external model, and then copy and paste it into the MasterSim. Link up the new subsystem just like the old one and then promptly delete the old link from existence. Congratulations, the new subsystem is an actual part of the MasterSim now (not just a link) and it has inherited all of the configuration parameters of the MasterSim. Repeat for every externally-linked model. [NOTE: This entire process and the whole BlankConfig trick also might be negated or greatly simplified through the use of Configuration Reference. Look into for the future.]

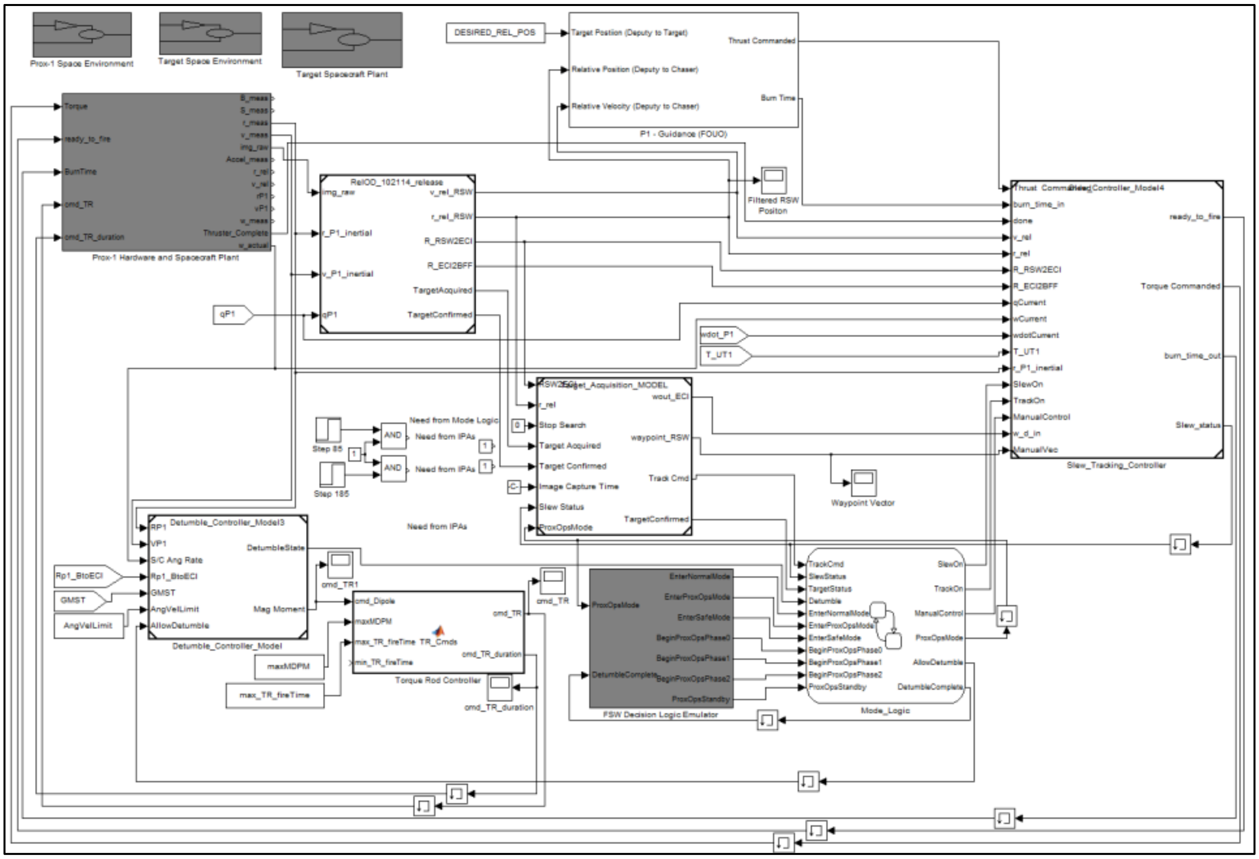
Generation

Steps:

- Highlight every piece of the MasterSim that will not be autocoded (Environment need not be included), right click, and select 'Create Subsystem From Selection'. Do the same for what will be autocoded. You should have two main subsystems now. Right click the subsystem to be autocoded, scroll over 'C/C++ Code' and select 'Build This Subsystem'. The actual autocoding process should now begin. Confirm the decision to build the code on any screens that may appear and let the Coder do its work.



- ➢ In all MasterSim versions starting at version 4.0 and later, it is easy to tell which subsystems need to be autocoded and which ones don’t, because the ones that should not be autocoded are grayed out.



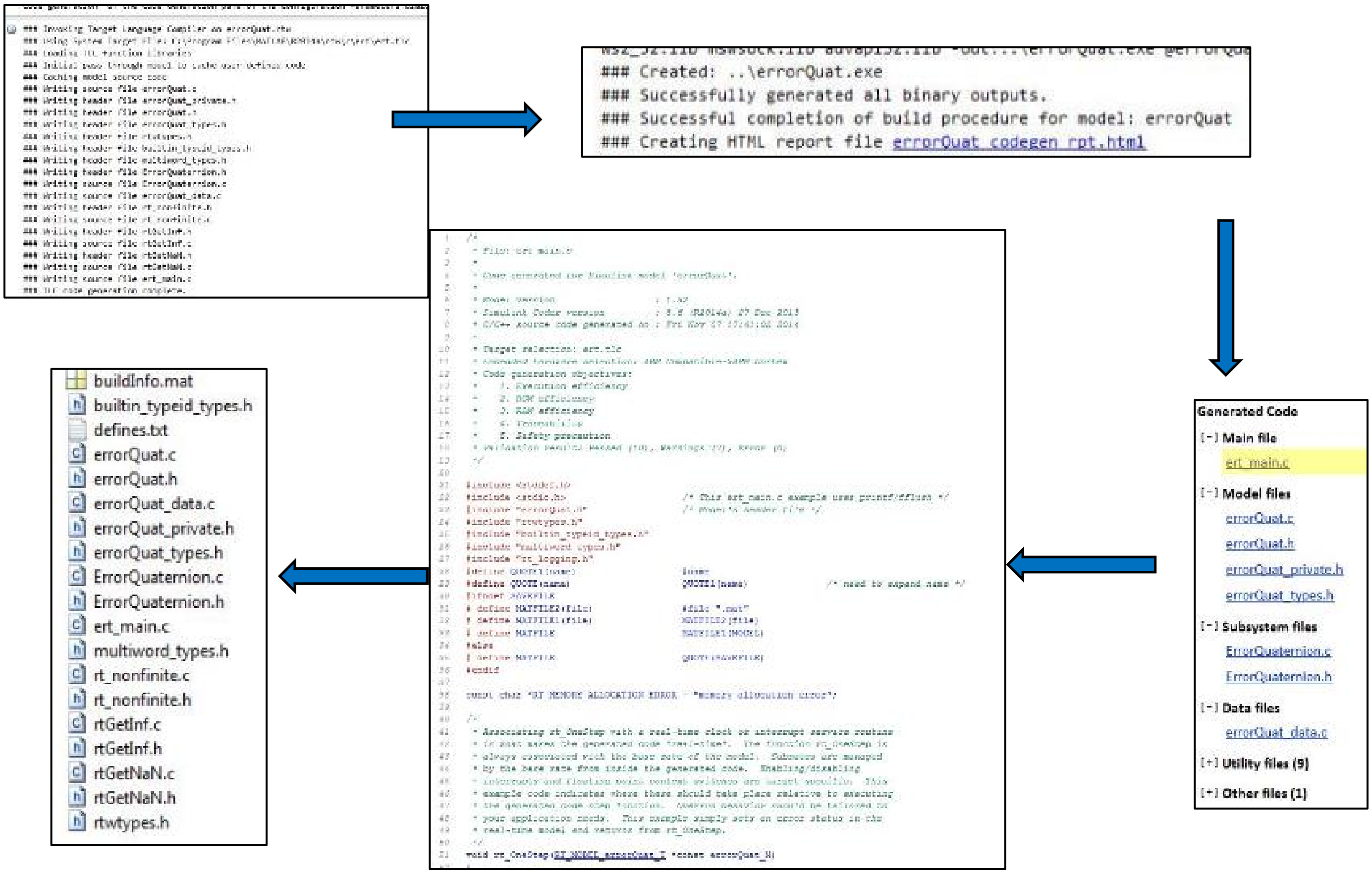
- If the code successfully builds then congratulations! The hardest part is over and the Simulink model has been converted to C code. All that remains is to check that the C code is functional. If it doesn't then try to comprehend the error that the Coder is giving you and work out the problem. Make sure that you followed each of the previous steps.

- A possible error that may occur is that an extrinsic function was used in the simulation. In brief, an extrinsic function is one that requires MATLAB to run, and therefore cannot be autocoded. If this occurs, try to find an autocodable equivalent to the extrinsic function. If not can be found then you must take up the job of coding the function into C yourself. Import this C function into MATLAB and replace the old extrinsic function where necessary. Hopefully, this will not happen too often. (It really takes the 'auto' out of 'autocoding' and is a fairly difficult problem to solve.)

Steps:

- While most of the Compilation step actually falls outside the realm of autocoding, you can manually put in preprocessor commands into the C code in order to eyeball and roughly make sure that the outputted values from the C code are reasonably accurate. Use the preprocessor commands to set up different debugging modes.

Each mode will let the program print out the selected variable at each time step. The value acquired from the program printing can then be compared to the data created during the Simulation, and if they seem to match then the program is probably on the right track. This is slightly advanced and requires some basic C concepts to understand. It also is not a very efficient use of time. As a result, this method is more or less summarized without graphical representation or in-depth explanation, which would not add much value for those who understand C and would most likely confuse those who do not know what is going on. [NOTE: Future Documentation Guides will go into the Compilation step of Autocoding when it is more refined. Currently, it is in its infancy. Also, for the future a more statistically sound way of doing this should be implemented. A C program that automatically checks C values versus Simulink values would be ideal. Until then, this should suffice.]



```
### Invoking Target Language Compiler on errorQuat.rtw
### Using System Target File: C:\Program Files\MATLAB\R2014a\rtw\c\ert\ert.tlc
### Loading TLC function libraries
### Initial pass through model to cache user defined code
### Caching model source code
### Writing source file errorQuat.c
### Writing header file errorQuat_private.h
### Writing header file errorQuat.h
### Writing header file errorQuat_types.h
### Writing header file rtwtypes.h
### Writing header file builtin_typeid_types.h
### Writing header file multiword_types.h
### Writing header file ErrorQuaternion.h
### Writing source file ErrorQuaternion.c
### Writing source file errorQuat_data.c
### Writing header file rt_nonfinite.h
### Writing source file rt_nonfinite.c
### Writing header file rtGetInf.h
### Writing source file rtGetInf.c
### Writing header file rtGetNaN.h
### Writing source file rtGetNaN.c
### Writing source file ert_main.c
### TLC code generation complete.
```

```
ws2_32.lib mswsock.lib advapi32.lib -out:..\errorQuat.exe @errorQua
### Created: ..\errorQuat.exe
### Successfully generated all binary outputs.
### Successful completion of build procedure for model: errorQuat
### Creating HTML report file errorQuat_codegen_rpt.html
```

```
/*
 * File: ert_main.c
 *
 * Code generated for Simulink model 'errorQuat'.
 *
 * Model version                  : 1.52
 * Simulink Coder version         : 8.6 (R2014a) 27-Dec-2013
 * C/C++ source code generated on : Fri Nov 07 17:43:02 2014
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: ARM Compatible->ARM Cortex
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. ROM efficiency
 *    3. RAM efficiency
 *    4. Traceability
 *    5. Safety precaution
 * Validation result: Passed (10), Warnings (2), Error (0)
 */

#include <stddef.h>
#include <stdio.h>                     /* This ert_main.c example uses printf/fflush */
#include "errorQuat.h"                 /* Model's header file */
#include "rtwtypes.h"
#include "builtin_typeid_types.h"
#include "multiword_types.h"
#include "rt_logging.h"
#define QUOTE1(name)                   #name
#define QUOTE(name)                    QUOTE1(name)    /* need to expand name */
#ifndef SAVEFILE
# define MATFILE2(file)                #file ".mat"
# define MATFILE1(file)                MATFILE2(file)
# define MATFILE                       MATFILE1(MODEL)
#else
# define MATFILE                       QUOTE(SAVEFILE)
#endif

const char *RT_MEMORY_ALLOCATION_ERROR = "memory allocation error";

/*
 * Associating rt_OneStep with a real-time clock or interrupt service routine
 * is what makes the generated code "real-time".  The function rt_OneStep is
 * always associated with the base rate of the model.  Subrates are managed
 * by the base rate from inside the generated code.  Enabling/disabling
 * interrupts and floating point context switches are target specific.  This
 * example code indicates where these should take place relative to executing
 * the generated code step function.  Overrun behavior should be tailored to
 * your application needs.  This example simply sets an error status in the
 * real-time model and returns from rt_OneStep.
 */
void rt_OneStep(RT_MODEL_errorQuat_T *const errorQuat_M)
{
```

Technical Memorandum
Original: April 8, 2015
Updated: March 10, 2017

TO: Air Force Research Lab, University Nanosatellite Program

FROM: Peter Schulte (GN&C Lead), Jacob Sussman (GN&C), Nolan Coulter (GN&C), Matthew Krumwiede (FSW)

SUBJECT: Prox-1 Autocoding Process, Verification, and Software Integration

The Simulink design of the Guidance, Navigation, and Control (GN&C) algorithms for Prox-1 is autocoded into C/C++ using Simulink Coder for integration with Flight Software (FSW). Some modifications are made to ensure the models are codable, such as avoiding the use of incompatible MATLAB functions. This process of GN&C algorithm development and integration in Simulink and later autocoding into flight software has been pioneered by the Orion spacecraft team at NASA's Johnson Space Center in Houston, Texas [1].

1. Autocoding Process

The process for autocoding from a Simulink master simulation to C code is described in this section based on work completed by Prox-1 team members Jacob Sussman and Meet Patel. This process is illustrated at a high level by the flowchart in Figure 1.

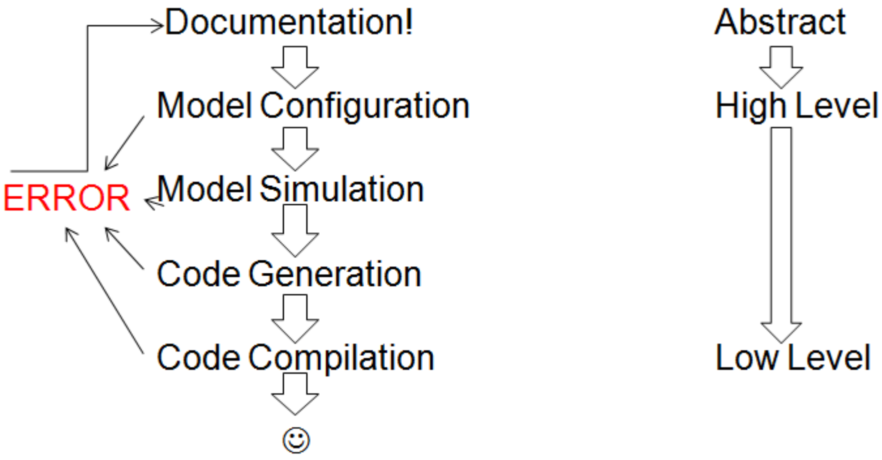


Figure 1: Autocoding process flowchart

The first step in this process is documentation, which is critical for capturing detailed instructions and lessons learned such as best practices and how to deal with common errors. Next is model configuration, which involves setting a multitude of parameters within the Simulink model to be autocoded. Proper model configuration allows for smoother model simulation and code generation. Also, these configuration parameters can optimize the resulting generated code to run on a specific embedded processor. Once the proper configuration settings are determined, they can be saved as a separate file and maintained using configuration control. Developers can then apply these standard configuration settings to any model by importing that file in Simulink.

The next step is model simulation. A Simulink model such as the Prox-1 GN&C master simulation (MasterSim) shown in Figure 2, cannot be autocoded if it does not run properly in simulation. In the version of the MasterSim shown above, the Simulink diagram includes many GN&C components such as the Relative Orbit (RelOD) filter, Artificial Potential Function (APF) guidance, and torque rod (TR) controller. Gray blocks are only used for simulation purposes, while white blocks will be autocoded into GN&C flight software (FSW). For configuration control, most of the GN&C component blocks are integrated into the master simulation using model reference blocks. These allow each component to be saved as a separate Simulink file that can be integrated into multiple master simulations.

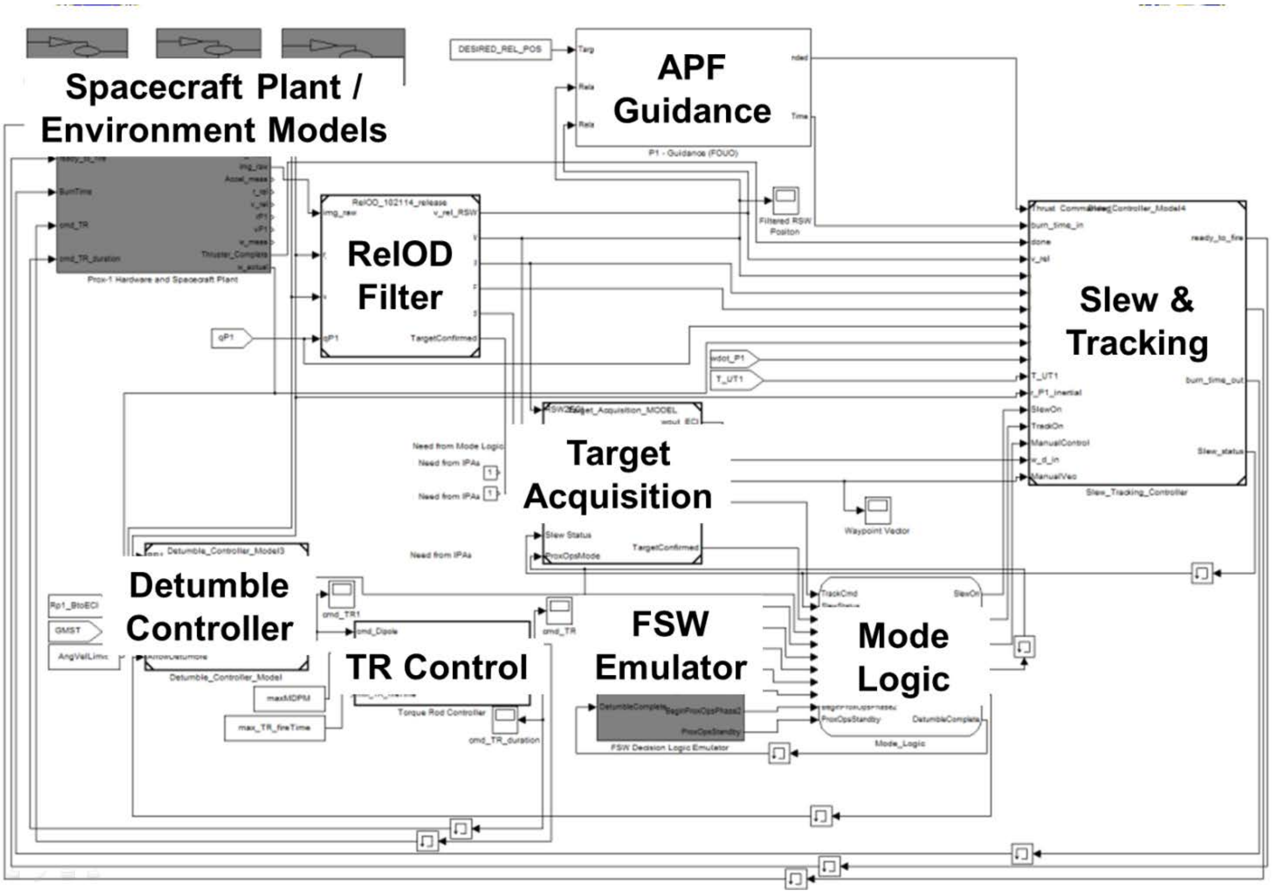


Figure 2: A version of the Prox-1 6DOF “master simulation”

After it is verified that the simulation model can run and provides the desired outputs, any blocks included from separate files as model references are copied and pasted into a single Simulink model to simplify the autocoding process. The model is then reconfigured so that all GN&C blocks to be autocoded are combined into a single monolithic GN&C algorithm block as shown in Figure 3. The spacecraft plant and environment models remain in separate blocks and will not be autocoded. Before autocoding of the GN&C algorithms begins, the reconfigured simulation is run again to ensure that no changes in performance or outputs have been introduced by the reconfiguration process.

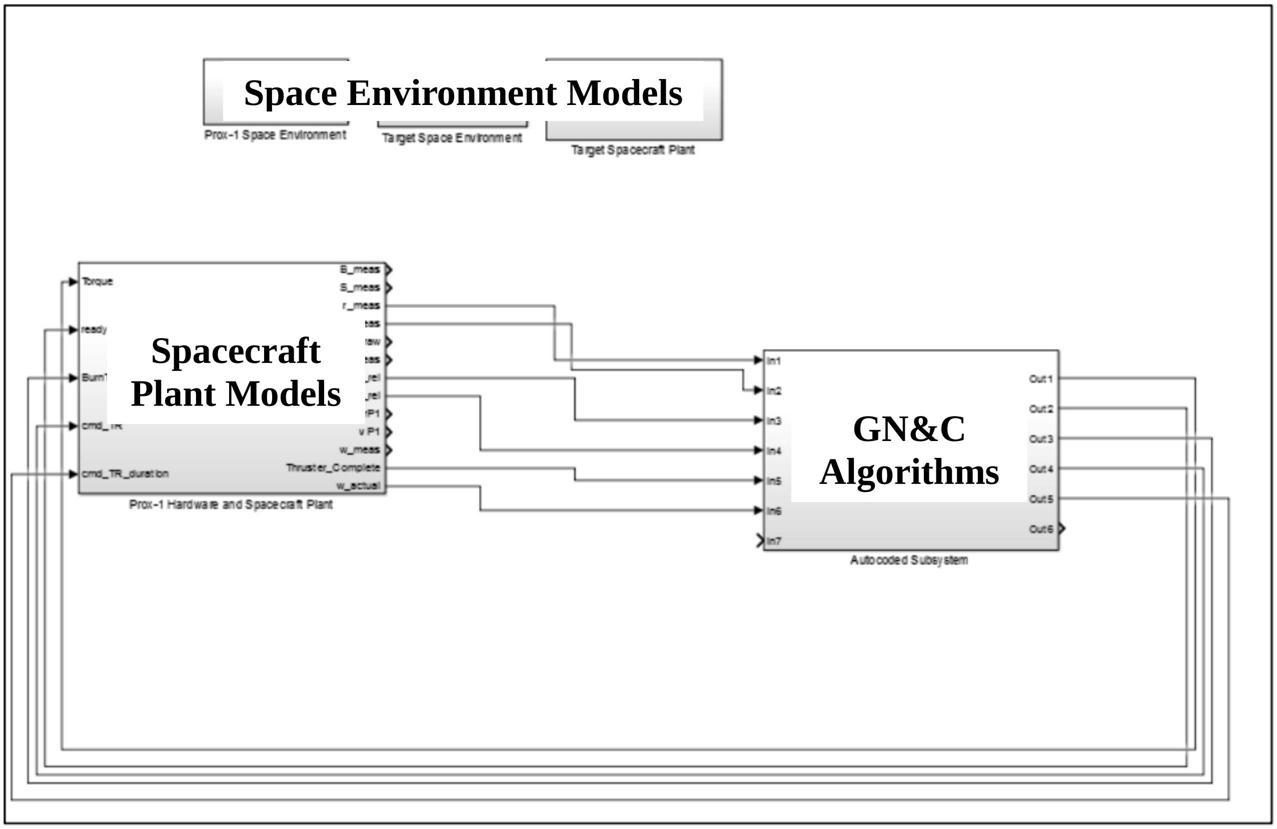


Figure 3: Master sim model prepared for autocoding by creating a single GN&C block

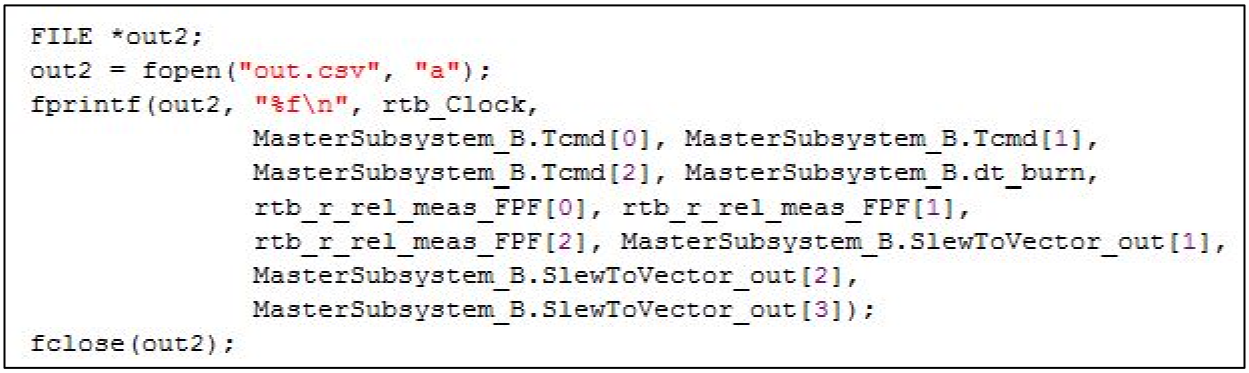
Once all of these changes have been made, the model is ready for code generation. Many errors can occur during this stage, and each error must be understood and corrected. An error appendix detailing common errors and how to fix them is located at the end of this document. After the code is generated, it must be compiled to run on the BeagleBoard XM flight computer. Finally, the compiled code should be run to verify that the outputs match those of the Simulink model.

2. Autocoding Verification

At the beginning of the Spring 2015 semester, Prox-1 team member Nolan Coulter was assigned the task to develop a process to verify the outputs created by the autocoded C version of the GN&C algorithms matched the corresponding outputs given by the Simulink program in MATLAB. This would allow the team to verify that the autocoded program (from here on referred to as just autocode) performs properly and there are no errors in the autocode. Our team used this verification process with several variations of the MasterSim Simulink program. With this verification process, the team was able to conclude that the outputs from the autocode matched the outputs produced by the Simulink program within a very small degree error.

2.1 Verification Process for "MATLAB-logging" Autocode

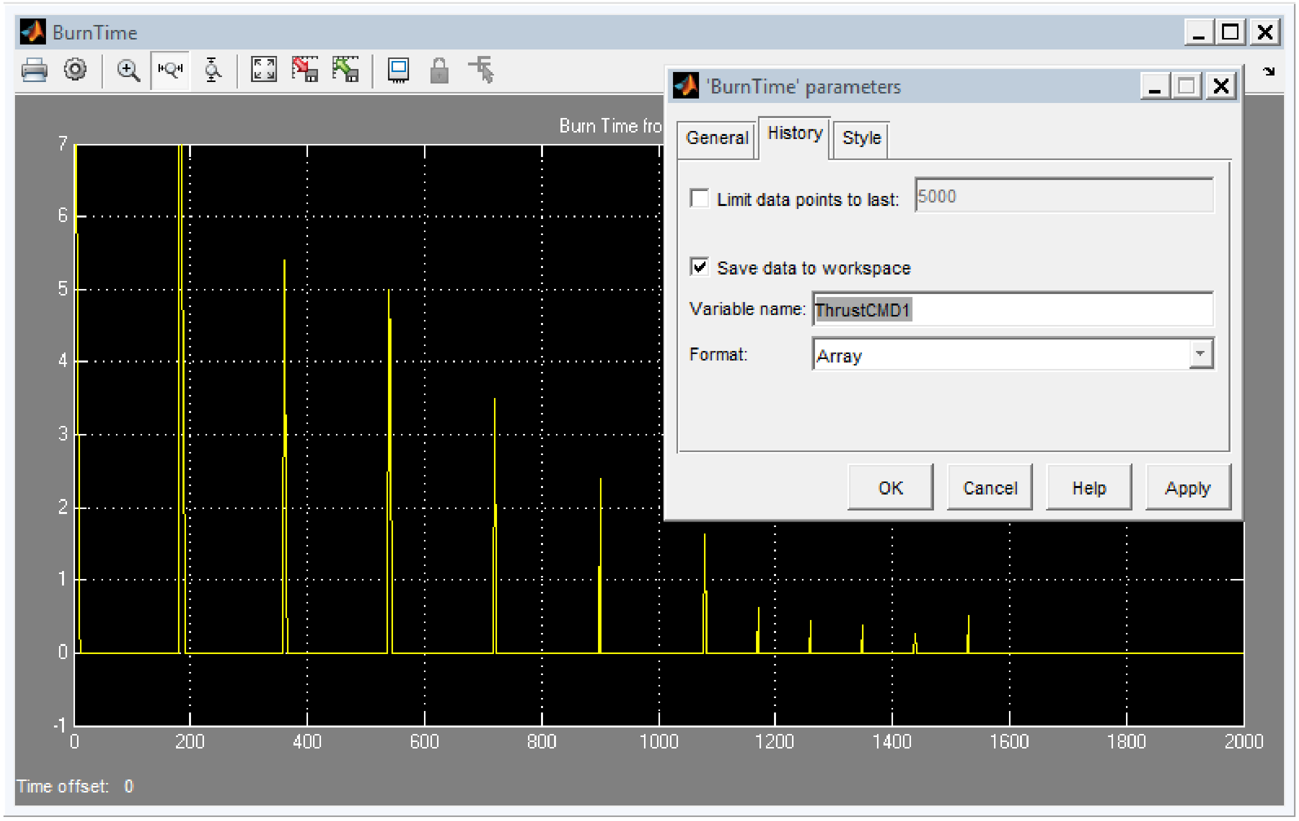
The process used to conclude that the autocode outputs corresponded with the Simulink outputs involved a series of steps that took the autocode and compared it to the Simulink program. When working with the autocode, the first step to be able to compare the data with the Simulink data is to give the proper commands to the autocode to record and log the outputs. After being autocoded with default settings, the program performs the basic calculations required to produce the outputs for each timestep; however, the autocode by default does not record each timestep. In the autocode, there is an "xxx_step" method that is used to update the outputs after each timestep. Near the end of this method, it updates the memory variables needed to perform the next timestep's calculations. These variables are needed to verify the outputs. At the end of this method, after the memory updates, the code shown in Figure 4 was added to record the data after each timestep. This code creates an output file called "out.csv" in csv format that logs the variables of interest.



```
FILE *out2;
out2 = fopen("out.csv", "a");
fprintf(out2, "%f\n", rtb_Clock,
              MasterSubsystem_B.Tcmd[0], MasterSubsystem_B.Tcmd[1],
              MasterSubsystem_B.Tcmd[2], MasterSubsystem_B.dt_burn,
              rtb_r_rel_meas_FPF[0], rtb_r_rel_meas_FPF[1],
              rtb_r_rel_meas_FPF[2], MasterSubsystem_B.SlewToVector_out[1],
              MasterSubsystem_B.SlewToVector_out[2],
              MasterSubsystem_B.SlewToVector_out[3]);
fclose(out2);
```

Figure 4: C-code to generate output file

After this output file is created, the file can then be used to graph the variables of interest over time or be directly uploaded into MATLAB for comparison. Once the required data is put into this format, the outputs of the autocode can be compared to the outputs from the Simulink program by going into the individual output arrays in MATLAB which are created after running the Simulink simulation. After running the simulation, the variables needed for comparing can be found in the MATLAB Workspace. The correct output array can be determined by noting the name of the output array MATLAB uses to save the values of the variable to the Workspace. The name is found under the “History” tab in the settings window of the variable’s corresponding scope in Simulink. Figure 5 depicts the scope of the BurnTime variable in Simulink, while Figure 6 provides a view of the MATLAB output arrays and the ThrusterCMD1 array (which contains the value of BurnTime) used to verify the autocode BurnTime output. The left column of the ThrusterCMD1 array shows the simulation time in seconds and the right column shows the BurnTime command, also in seconds.



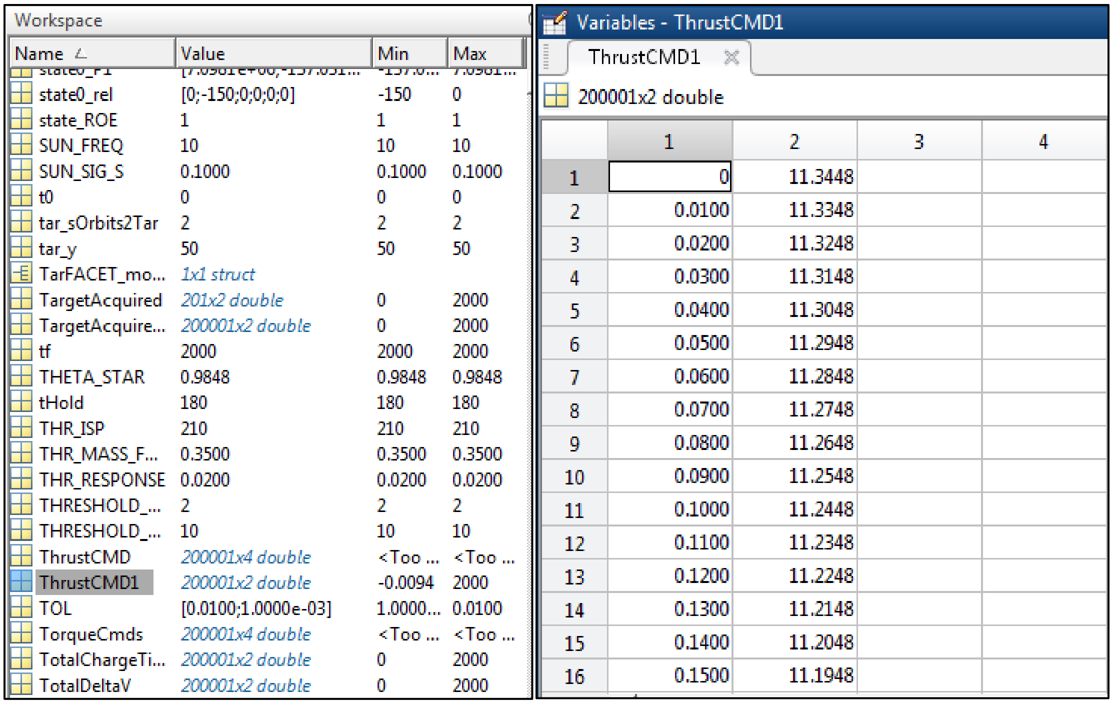
□ Limit data points to last: 5000

☑ Save data to workspace

Variable name: ThrustCMD1

Format: Array

Figure 5: View of Simulink “Scope” for the BurnTime variable



Workspace

Name	Value	Min	Max
state0_FI	[7.0981e+06;-137.031...	-137.0...	7.0981...
state0_rel	[0;-150;0;0;0;0]	-150	0
state_ROE	1	1	1
SUN_FREQ	10	10	10
SUN_SIG_S	0.1000	0.1000	0.1000
t0	0	0	0
tar_sOrbits2Tar	2	2	2
tar_y	50	50	50
TarFACET_mo...	1x1 struct		
TargetAcquired	201x2 double	0	2000
TargetAcquire...	200001x2 double	0	2000
tf	2000	2000	2000
THETA_STAR	0.9848	0.9848	0.9848
tHold	180	180	180
THR_ISP	210	210	210
THR_MASS_F...	0.3500	0.3500	0.3500
THR_RESPONSE	0.0200	0.0200	0.0200
THRESHOLD_...	2	2	2
THRESHOLD_...	10	10	10
ThrustCMD	200001x4 double	<Too ...	<Too ...
ThrustCMD1	200001x2 double	-0.0094	2000
TOL	[0.0100;1.0000e-03]	1.0000...	0.0100
TorqueCmds	200001x4 double	<Too ...	<Too ...
TotalChargeTi...	200001x2 double	0	2000
TotalDeltaV	200001x2 double	0	2000

Variables - ThrustCMD1

ThrustCMD1

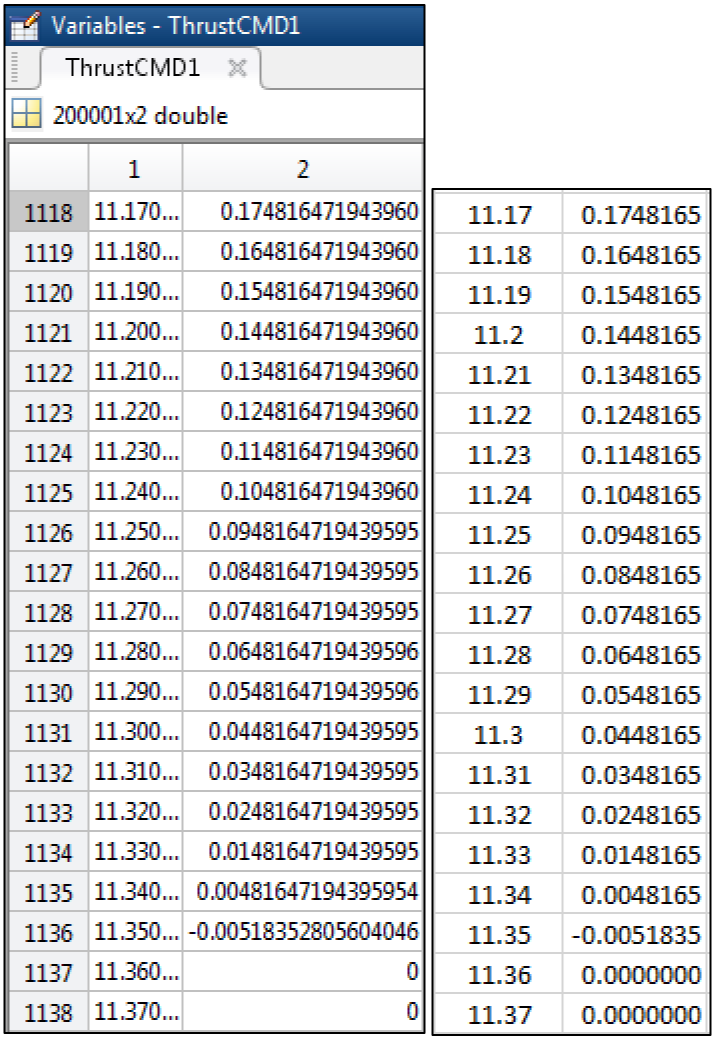
200001x2 double

	1	2	3	4
1	0	11.3448		
2	0.0100	11.3348		
3	0.0200	11.3248		
4	0.0300	11.3148		
5	0.0400	11.3048		
6	0.0500	11.2948		
7	0.0600	11.2848		
8	0.0700	11.2748		
9	0.0800	11.2648		
10	0.0900	11.2548		
11	0.1000	11.2448		
12	0.1100	11.2348		
13	0.1200	11.2248		
14	0.1300	11.2148		
15	0.1400	11.2048		
16	0.1500	11.1948		

Figure 6: Table of MATLAB Output Arrays (left) and ThrustCMD1 Output Array (right)

If the data in the MATLAB output array is equivalent to the data in the output .csv file from the autocode, then the autocode performs as expected and contains no errors. This comparison can be seen in Figure 7.

]



	1	2
1118	11.170...	0.174816471943960
1119	11.180...	0.164816471943960
1120	11.190...	0.154816471943960
1121	11.200...	0.144816471943960
1122	11.210...	0.134816471943960
1123	11.220...	0.124816471943960
1124	11.230...	0.114816471943960
1125	11.240...	0.104816471943960
1126	11.250...	0.0948164719439595
1127	11.260...	0.0848164719439595
1128	11.270...	0.0748164719439595
1129	11.280...	0.0648164719439596
1130	11.290...	0.0548164719439596
1131	11.300...	0.0448164719439595
1132	11.310...	0.0348164719439595
1133	11.320...	0.0248164719439595
1134	11.330...	0.0148164719439595
1135	11.340...	0.00481647194395954
1136	11.350...	-0.00518352805604046
1137	11.360...	0
1138	11.370...	0

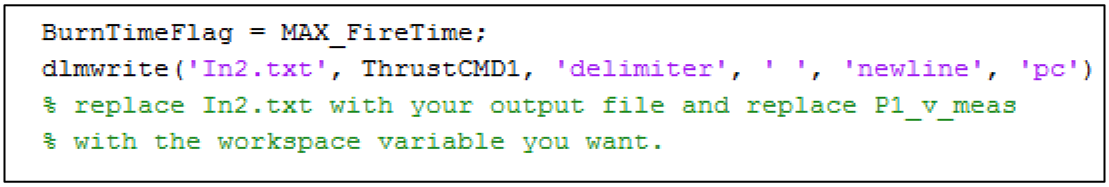
11.17	0.1748165
11.18	0.1648165
11.19	0.1548165
11.2	0.1448165
11.21	0.1348165
11.22	0.1248165
11.23	0.1148165
11.24	0.1048165
11.25	0.0948165
11.26	0.0848165
11.27	0.0748165
11.28	0.0648165
11.29	0.0548165
11.3	0.0448165
11.31	0.0348165
11.32	0.0248165
11.33	0.0148165
11.34	0.0048165
11.35	-0.0051835
11.36	0.0000000
11.37	0.0000000

Figure 7: Comparison between the BurnTime output as seen in the MATLAB workspace (left) and the BurnTime output as seen from the autocode .csv output file (right)

2.2 Verification Process for non-"MATLAB-logging" Autocode

The process described in Section 2.1 was developed using autocode, that when being autocoded, used the option "include MATLAB logging." This option in the autocoding process created a library of the variables and their values that the resultant autocode could access and use to initialize the timestep. This option was not used in more recent versions of the autocode, which means that additional steps are required to acquire the needed data from the autocode to compare with Simulink.

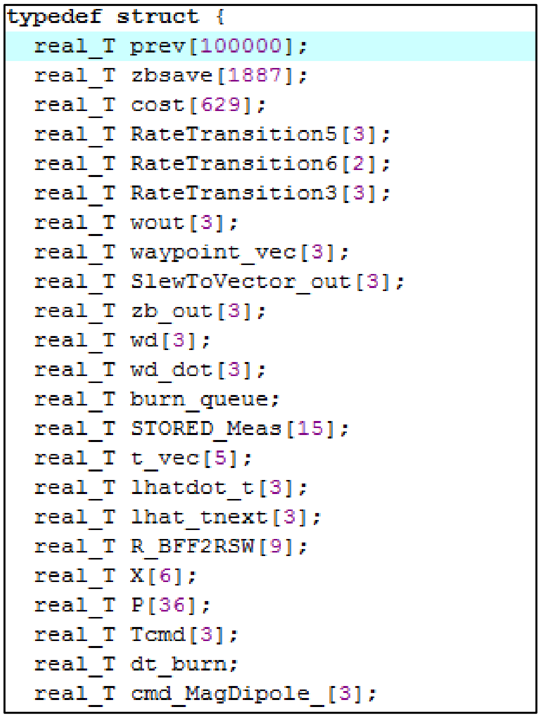
In order to initialize the timestep method which then calculates the required data at each subsequent timestep, the initialization data from MATLAB must first be recorded manually. This requires observing the Simulink simulation file and the autocode to see which input variables are necessary to begin the timestep method. After listing the needed variables, the code found in Figure 8 must be added to the embedded MATLAB code within Simulink corresponding to the generation of that variable. After running the simulation in Simulink with this added code, the variables will be output as a .csv file by MATLAB.



```
BurnTimeFlag = MAX_FireTime;
dlmwrite('In2.txt', ThrustCMD1, 'delimiter', ' ', 'newline', 'pc')
% replace In2.txt with your output file and replace P1_v_meas
% with the workspace variable you want.
```

Figure 8: MATLAB code to record a required variable (in this example BurnTimeFlag)

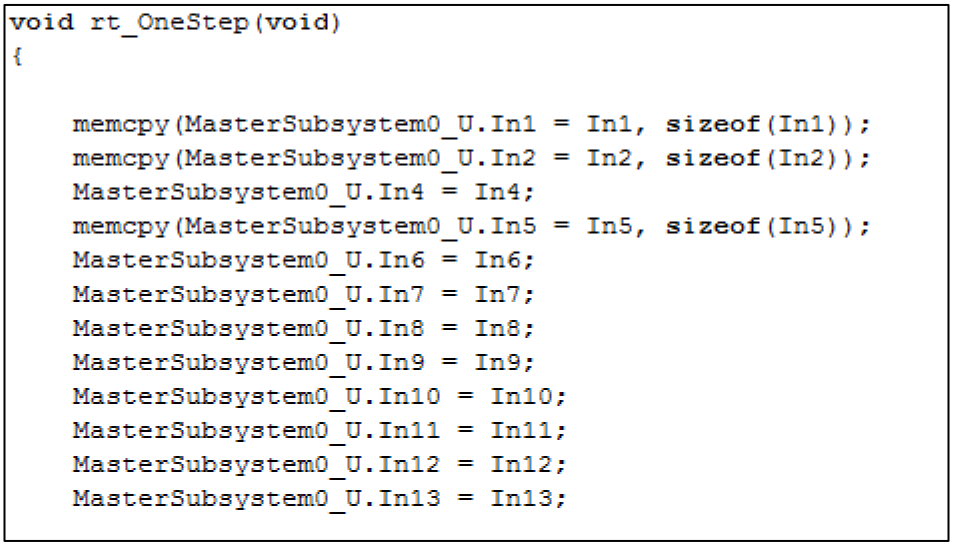
Once the necessary variables' data is logged into separate .csv files, changes in the autocode must be made. Going into the autocode's header file, the variables previously logged manually must be defined according the same dimensions found in the MATLAB output array. This header file initializes and defines the structure of the variables used throughout the autocoded functions. The code required to define these structures is shown in Figure 9.



```
typedef struct {
  real_T prev[100000];
  real_T zbsave[1887];
  real_T cost[629];
  real_T RateTransition5[3];
  real_T RateTransition6[2];
  real_T RateTransition3[3];
  real_T wout[3];
  real_T waypoint_vec[3];
  real_T SlewToVector_out[3];
  real_T zb_out[3];
  real_T wd[3];
  real_T wd_dot[3];
  real_T burn_queue;
  real_T STORED_Meas[15];
  real_T t_vec[5];
  real_T lhatdot_t[3];
  real_T lhat_tnext[3];
  real_T R_BFF2RSW[9];
  real_T X[6];
  real_T P[36];
  real_T Tcmd[3];
  real_T dt_burn;
  real_T cmd_MagDipole_[3];
```

Figure 9: Code to define structure of input variable array in autocoded functions

Once this data is given a defined structure, it can then be assigned to the proper variable found in the autocode. Figure 10 shows the code required to assign the data to the corresponding variable in the autocode's main file. Once the necessary variables are defined, the autocode functions should run and produce the outputs. These files can then be compared to the MATLAB output arrays in a similar fashion as the "MATLAB-logging" capable process.



```
void rt_OneStep(void)
{

    memcpy(MasterSubsystem0_U.In1 = In1, sizeof(In1));
    memcpy(MasterSubsystem0_U.In2 = In2, sizeof(In2));
    MasterSubsystem0_U.In4 = In4;
    memcpy(MasterSubsystem0_U.In5 = In5, sizeof(In5));
    MasterSubsystem0_U.In6 = In6;
    MasterSubsystem0_U.In7 = In7;
    MasterSubsystem0_U.In8 = In8;
    MasterSubsystem0_U.In9 = In9;
    MasterSubsystem0_U.In10 = In10;
    MasterSubsystem0_U.In11 = In11;
    MasterSubsystem0_U.In12 = In12;
    MasterSubsystem0_U.In13 = In13;
```

Figure 10: Block of code that assigns autocode variables to input data arrays

2.3 Error in Data Comparisons

While comparing the autocode outputs with the Simulink outputs, it is important to note that the two types of data do not match exactly. This is largely due to rounding errors in the .csv file. Looking at the .csv file containing the data of the autocode outputs, for certain variables such as the relative velocity vector, the values appear as -0.00000X while the corresponding value in the MATLAB output array is has more significant figures and a more precise value. The .csv file rounds these excessive significant figures. However, these small rounding errors can be disregarded since the two output files agree to several significant figures.

2.4 Autocoding Verification Conclusions

The process developed to verify that the autocode outputs were equivalent to the Simulink and MATLAB outputs was successful. The outputs from the autocode matched the outputs from Simulink simulation within the aforementioned degree of error. However, it is important to distinguish that this verification process is an open-loop system and does not test a closed-loop dynamic model. The inputs fed into the autocode are simply saved values from the Simulink code. This means that at each timestep, there is an already defined input used to calculate the corresponding output. The output from the previous timestep is not fed back into the next timestep through spacecraft plant and dynamics models because these models are not autocoded. If the output from a previous timestep was then used to calculate the input for the next, this would be a closed loop system. In the process described in this section, the inputs are independent of the outputs from the previous timestep. With a closed loop process, the necessary Spacecraft Hardware Plant and Environment Models would need to be autocoded in addition to the GN&C algorithms. Although a closed-loop test was not performed, it was verified that the C code provides the same results as the Simulink models at each individual open-loop timestep.

3. Integration with Flight Software

After the C code has been generated for use on the BeagleBoard, it must be integrated with FSW. In the current understanding of the Prox-1 GN&C and FSW teams, the GN&C code will be integrated as a monolithic sub-routine that is called during each timestep. This process is illustrated in Figure 11, which shows that external variables are collected by FSW from various sensors (1) and sent into the GN&C code (2), which then operates in various modes. GN&C returns commands to FSW (3), which then distributes those commands to actuator hardware such as the Propulsion and Attitude Determination and Control Subsystem (ADCS) microcontrollers. In (2) and (3) FSW and GN&C will also exchange mode logic variables, such as a command to enter ProxOpsMode from the ground or an indication from GN&C that a Guidance command has been successfully executed.

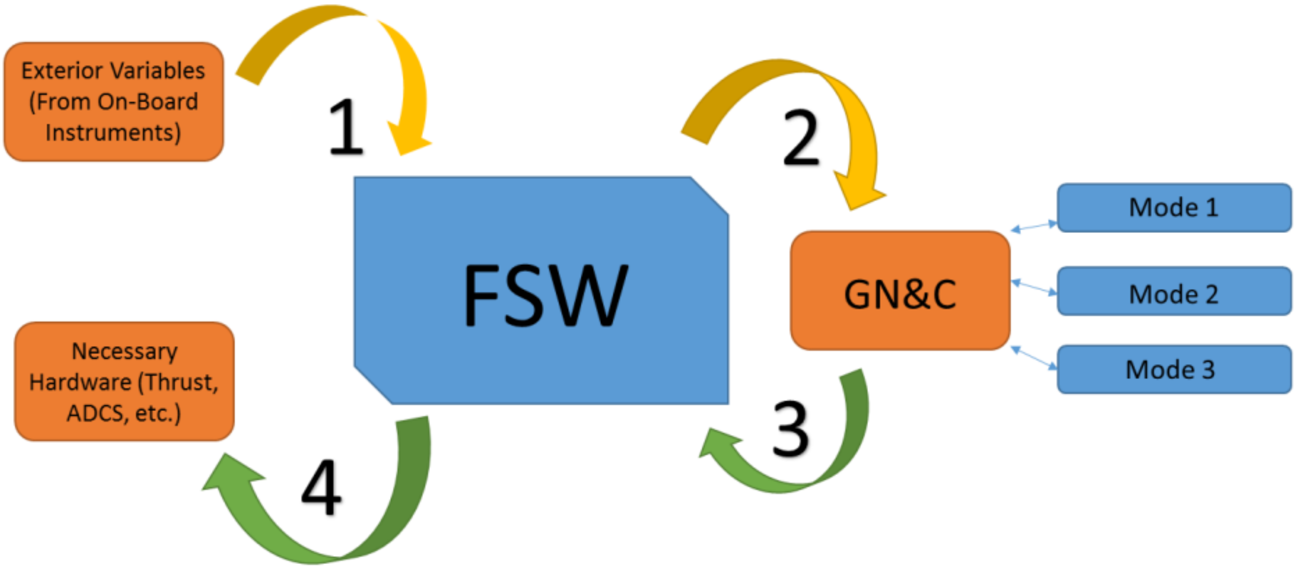

Exterior Variables (From On-Board Instruments)
1
2
FSW
GN&C
Mode 1
Mode 2
Mode 3
Necessary Hardware (Thrust, ADCS, etc.)
3
4

Figure 11: GN&C/FSW interface illustration

(Original Spring 2015 text; applicable when Prox-1 was using the Core Flight Executive)

The actual mechanism of integration between GN&C and FSW involves the Core Flight Executive (CFE), which is code produced by NASA that greatly facilitates in the production of flight software [2]. Once the C code is generated from Simulink, all files are transferred to the Prox-1 CFE github repository. Under the *apps* directory, which is located in *FSW/cFE*, a new *gnc_app* directory is created, and all the auto-coded files are relocated here. The structure of the *gnc_app* directory is shown in Figure 12.

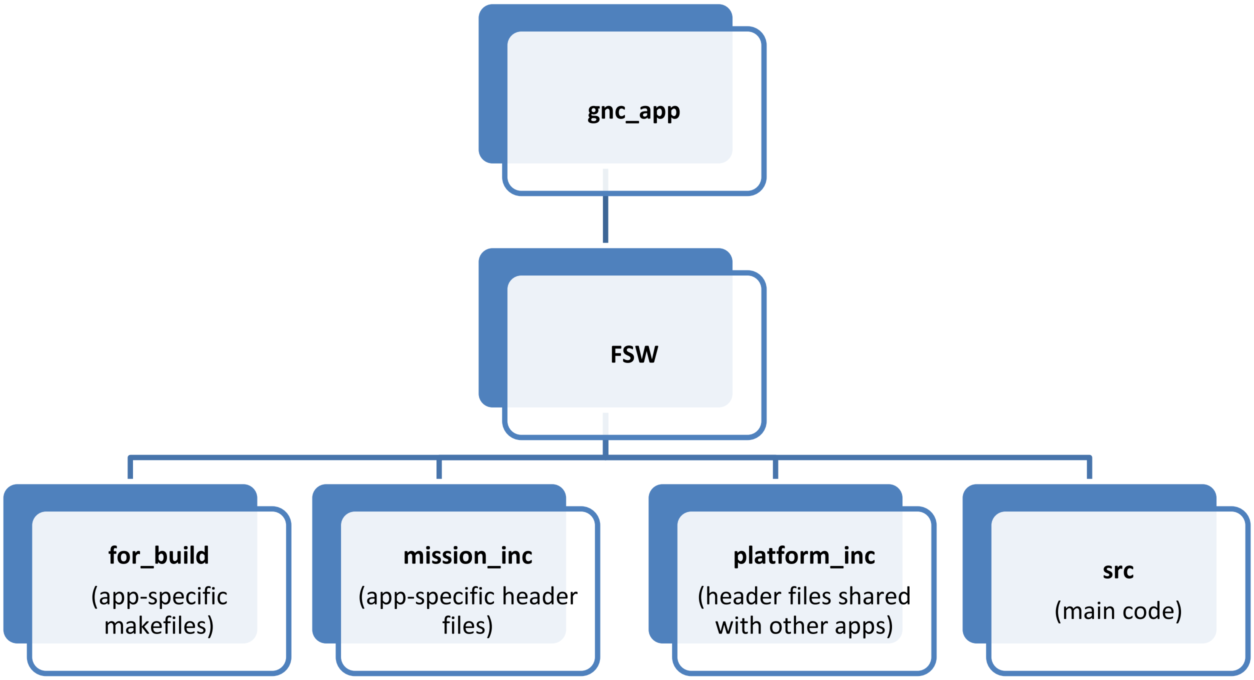


Figure 12: gnc_app directory structure

Most of the code is transferred to the *src* directory, while the header files that have important information for other apps are placed in the *platform_inc* directory. Once the files are relocated, a new C file is created, *gnc_app.c*, along with a header file, *gnc_app.h*. The new C file follows CFE protocol for creating an app. The main function of *gnc_app.c* listens for messages from other applications (mainly Prox_app, the master application for Prox-1 FSW). The messages will contain a data type, along with a command ID that tells GN&C what input the data corresponds to (message ID's are defined in the *platform_inc* subdirectory). The main function also has logic that determines if all thirteen inputs necessary for GN&C to run have been received. If not, the main function continues on a loop.

Once all the inputs are received, *gnc_app.c* calls *MasterSubsystem0.c*, which is the main auto-coded C file from Simulink. The Master Subsystem takes the inputs and calculates an output structure. The output structure is forwarded back to the main application of *gnc_app.c*, and then sent via a software bus to other applications.

To determine how to compile the application, *MasterSubsystem0.c* and its dependencies were all compiled outside of CFE, independently, alongside a tester file *ert_main.c*. It was discovered that the C flag *–lm* is required for compilation. With this information, compilation of gnc_app with CFE was very straightforward. The makefile within the *for_build* subdirectory was modified to include all the necessary dependencies and the *–lm*

C flag necessary for compilation. Everything else was taken care of by the CFE, so that when the whole of Flight Software was compiled, gnc_app compiled as well.
Testing of the FSW/GNC integration will be performed through an external daemon server that poses as a serial device interacting with CFE. Since the flight computer will not receive actual relevant data from hardware until it is in orbit, the daemon simulator is necessary to test the performance of GN&C in an open-loop manner. The daemon server will obtain "fake" input data from text files. The fake inputs are collected from the original Simulink code, which has a hardware simulation module. These fake inputs are fed through the serial port one by one, interacting first with the serial app, then going through the main Prox_app, then finally making their way to *gnc_app*. The first test will involve making sure that the autocoded GN&C function does not execute the first iteration until it receives *all* thirteen inputs. The next test will involve running GN&C through a certain number of iterations while comparing the output structures to the outputs in the original Simulink model outputs using a process similar to the one presented in Section 2.

Finally, a closed-loop test will be performed using hardware-in-the-loop simulation with the FSW on the BeagleBoard connected to the Spacecraft Plant and Environment models in Simulink via MATLAB's xPC Target toolbox. This test will verify that the completed *gnc_app* within the FSW code performs in the same manner as the GN&C algorithms in Simulink using the same Simulink simulation plant and environment models.

(Updated Spring 2017 text after Prox-1 abandoned cFE in favor of a custom executive)

The algorithms, initially after the autocoding process, are stored in a main C file as hundreds of individual functions. Within this C file, four main functions (here called s*tep functions*) handle the operation of the algorithms, and thus act as the main functions of the autocode. For each time step, a step function is called, with the hardware variables (sun sensor data, accelerometer data, etc.) as inputs. The step function then determines which algorithms to run, based on the elapsed time of the mission. All four step functions must work together in unison at any given time step, such that the entire system resembles the Simulink model as closely as possible.

The goal of the integration process is to retrieve the hardware variables from flight software and pass them into all four step functions synchronously at each time interval. The outputs of the model must then be read and sent to the hardware as soon as possible, ideally within the same time step. To facilitate this integration process, a C file was created in the flight code directory that was responsible for handling the GN&C step functions. Called *gnc.c*, this file first defines all of the flight parameters that are not hard-coded into the model. This includes variables such as moments of inertia, mass, dimensions. It also includes variables which are subject to change. The file therefore contains a function, which, if called from the main flight code (due to a ground command), would change the value of the parameter and update the system. This allows for real-time parameter tuning.

After defining the parameters, the gnc.c file has an initial function which is called by the main flight code on startup. This function essentially creates the memory blocks which will store all of GN&C's data structures, and then populates them with the appropriate initial values. It also creates *mutexes*. Mutexes are useful for accessing the same variable from two different synchronous processes. If one process locks a mutex, it can perform a line of code that can't be accessed by another process until the mutex is unlocked. This avoids the essential problem of different threads trying to read and write to the same memory block at the same time (which would result in data corruption).

The "main" function of gnc.c is essentially a loop that runs every 0.1 seconds. During each iteration of the loop, the inputs for GN&C are retrieved. Most inputs are simply pulled from the flight code, which is already set up to retrieve hardware states via C&DH. Some inputs, however, are liable to change depending on how much time has passed, or on what state the satellite is in. For example, during detumble, certain inputs to GN&C must be manually set off (more information regarding the specifics of GN&C inputs can be found in other documentation). Therefore, the chief role of this main function is to use a series of logical checks, based on global state and mode variables, as well as the elapsed time of the mission, to determine the settings of certain inputs. After all inputs are obtained, they are stored inside of GN&C's global *inputs structure* using mutexes.

A separate function in gnc.c, called the *run loop*, runs separately from the main loop. It also runs every 0.1 seconds. At each iteration, the inputs structure is passed to each of the four step functions. The output of the step functions are stored in the *outputs structure*. The run loop then reads variables from the outputs structure and passes them to the main flight code, which are then sent to the appropriate hardware modules for command actuation.

In the main flight code, two pthreads are created at execution (after the startup routine). Pthreads are essentially sub-processes that can run independent of the main flight code. The first pthread calls the main loop of gnc.c, while the second pthread calls the run loop. At this point in the code, GN&C will start running in the background. This method of integration makes it much easier for engineers working on the main flight code to interface with GN&C. For example, each time Flight Software takes a hardware reading, it must simply store the corresponding values in the input structure of GN&C. The subprocess will take care of everything else. Likewise, to send an appropriate command to a piece of hardware, Flight Software must simply read the corresponding value from the output structure.

References

[1] Jackson, M.C. and Henry, J.R., "Orion GN&C Model Based Development: Experience and Lessons Learned," AIAA-2012-5036, *AIAA Guidance, Navigation, and Control Conference*, Minneapolis, Minnesota, August 2012.

[2] "Core Flight Executive (cFE)," Goddard Space Flight Center, Maryland, February 2015. [http://opensource.gsfc.nasa.gov/projects/cfe/index.php. Accessed 3/31/15.]

Appendix: Common Autocoding Errors

This section will show examples of common errors that occur during the autocoding process and how to fix them. When referenced, a "MasterSim" is a term for the Simulink file that is your master simulation, which contains all GN&C algorithms, spacecraft plant, and environment models. Errors shown list specific variable/function/model/path names but when debugging your own MasterSim the errors will be customized to your design.

Good rules of thumb to help with deal with errors:

1. Save often. You never know when MATLAB is going to throw a fit and you suddenly lose all of your unsaved work.
2. Make sure all Paths and proper initialization files are set up correctly.
3. Before the building phase, create a MasterSim which contains every component in one giant model as opposed to linking to external files. While this is a tedious process, it will prevent many future headaches.
4. Make sure your MasterSim can compile before you even try to build it. If it can't even run in MATLAB why would it run in C?

Initialization Errors

These errors occur because the initialization files (MATLAB scripts that must be run before the Simulink model is run) are not behaving properly.

```
"Undefined function or variable 'RP'.
Error in SpacecraftPlant_EnvironmentModel_Init (line 108)
P = (RE + RP)*(1+e_LS); %Semi-latus rectum (m)"
```

If there is more than one initialization file it is standard practice for one file to be the 'primary' initialization file. This primary .m file will then usually call all other necessary initialization files. When working under this design paradigm, it is therefore only necessary to call the primary file. If you run the wrong file mistaking it for the primary one or even run a secondary initialization file after already running the primary one, you may get this error. Secondary initialization files are written to assume they will be called by the primary file, meaning running them on their own will usually fail due lack of access to all of the defined primary variables.

```
"License checkout failed.
License Manager Error -4
Maximum number of users for Aerospace_Toolbox reached.
Try again later.
```

```
To see a list of current users use the lmstat utility or contact your License
Administrator.

Troubleshoot this issue by visiting:
http://www.mathworks.com/support/lme/R2014b/4

Diagnostic Information:
Feature: Aerospace_Toolbox
License path:
C:\Users\jsussman3\AppData\Roaming\MathWorks\MATLAB\R2014b_licenses;C
:\Program Files\MATLAB\R2014b\licenses\license.dat;C:\Program
Files\MATLAB\R2014b\licenses\network.lic
Licensing error: -4,132.

Error in HIL_6DOFMasterSim4_Init_TACTest (line 149)
q0_P1 = angle2quat(0,1,0); %initially point toward target”
```

The model was written with a Mathworks toolbox that you do not have access to. In this case it is the Aerospace Toolbox. To prevent this error from happening either make absolute sure everyone in the team is developing on the same environment or try to minimize use of toolbox functions in favor of writing custom versions yourself. To fix this example for instance you can rewrite line 149 as follows:

```
Q0_P1 = vector2quat([0;1;0]);
```

As long as you actually write the vector2quat function to be functionally equivalent to its toolbox’s counterpart, the initialization file should be able to run. This newly written function however is probably slower to execute than the toolbox version since it is not designed for efficiency by MathWorks. Be careful of replacing too many toolbox functions. It can make your code harder to debug and can slow development considerably. In an industry like Aerospace if you can’t afford a required software license, you probably can’t afford launching satellites in the first place.

```
“Undefined function 'ROT3' for input arguments of type 'double'.

Error in SpacecraftPlant_EnvironmentModel_Init (line 121)
r0_LS =
(ROT3(omega_LS)*ROT1(inc_LS)*ROT3(Omega_LS))'*[r0_LS*cos(f0
_LS);

Error in HIL_6DOFMasterSim5_Init_TACTest (line 268)
SpacecraftPlant_EnvironmentModel_Init;”
```

This error was caused by an initialization file which referenced a function located in a model not currently in the Path. To fix this, simply add the offending model to the Path and run the initialization file again.

Compilation Errors

These errors occur when attempting to run the MasterSim. Failure at this step prevents you from moving on towards code generation.

“Model 'Detumble_Controller_Model3_2012a' not found.”

In this case, a certain model (the Detumble Controller) is not connected to the MasterSim. To fix this, locate the model in MasterSim. It should be blocked in red. Double click on this block and point Simulink to where the model is located. When the popup box appears asking to add the file’s location to the Path click ‘Add to Path’.

“Error evaluating expression 't0' for 'StartTime' specified in the Configuration Parameters dialog for block diagram 'Detumble_Controller_Model3_2012a': Undefined function or variable 't0'.”

Did you run the initialization file before running the MasterSim? This error can occur if a user is so hasty to compile the MasterSim that they forget to run the initialization files first.

“Undefined function or variable 'v_desired'. The first assignment to a local variable determines its class. Function 'P1 - Guidance (FOUO)/p1_r2r_apf_v2 (FOUO)' (#135.3666.3675), line 93, column 17: "v_desired" Launch diagnostic report.”

When preparing a model for autocoding or simply when rearranging a model for aesthetic purposes in Simulink, you may accidentally break one of the MasterSim’s connection lines. To remedy this, locate where the undefined function/variable is in the context of the entire MasterSim and then trace the model to the offending location, fixing any broken connections lines along the way.

“Size mismatch (size [3 x 1] ~= size [3 x 3]). Function 'P1 - Guidance (FOUO)/p1_r2r_apf_v2 (FOUO)' (#135.2475.2529), line 55, column 32: `"((rDc_trans*P*rDc)-1)*Q*rCt-(rCt_trans*Q*rCt)*P*(-rDc)"`”

This error can occur if there is a broken connection just like the previous error. An undefined variable used in a matrix declaration can change a matrix’s size and throw off the rest of a model that anticipates a different size. This also could simply be a coding mistake, where the programmer messed up on declaring a matrix of the proper size.

“File…5update\HIL_6DOFMasterSim5_Init_TaCTest.m is not found in the current folder or on the MATLAB path.

To run this file, you can either change the MATLAB current folder or add its folder to the MATLAB path.”

This error will occur if your current working directory for MATLAB is not the one the MasterSim is located in or if the MasterSim’s folder is not in the Path. In the pop-up that follows this error click ‘Change to folder’ or ‘Add folder’ to remedy this.

“MATLAB has encountered an internal problem and needs to close.
The unsaved information you were working on may be lost. We are sorry for the inconvenience.
Click Never Send to disable sending information to MathWorks (saved in preference).
Click Send to send this information to Mathworks.
Click End Now to close MATLAB now.
Click Attempt to Continue to try to return briefly to MATLAB. You might be able to save your work.
Do not continue your MATLAB session after trying to save your work. Further operations are unreliable.
You must close and restart MATLAB in order for the program to operate correctly.
Click Details to see what will be sent to MathWorks if the Send button is clicked.”

Sometimes compilation will result in an ultimate error that forces you to shut down your working session and restart.

Clicking on Details usually results in a file telling you that an Access violation has been detected. This can have happened for a number of reasons. A simple example would be if you had a mismatched GOTO tag that didn’t have a proper pair. To prevent issues with GOTO tags in general you can bus every variable into the required model. This requires more effort but is generally safer and results in less issues during the building phase.

Building Errors

These are errors that occur when building the compiled model into C code. Most building errors occur from either not having MATLAB properly set up, not compiling the MasterSim in the form of one complete file, or not ridding your simulation of extrinsic functions.

“An installed compiler was not detected. Certain simulation modes, as well as host-based coder builds require that a compiler be installed. Please install one of the supported compilers for this release as listed at: http://www.mathworks.com/support/compilers/R2014b/win64.html MATLAB must be restarted after the compiler is installed.”

As the error tells you, the proper compiler is not installed. Make sure you have the right compiler and then continue.

“Configuration component 'RTWSystemTargetFile' of model 'HIL_6DOF_MasterSim4_APF_RelOD_TA' and configuration component 'RTWSystemTargetFile' of model 'ImageGen_SensorModel_v3' are not compatible. The error message returned by the comparison function is: The parameter setting for 'RTWSystemTargetFile' must be the same for all models in the model reference hierarchy”

If you try to build a model that contains externally-linked models there are a whole slew of problems that can occur. This is an example of MATLAB telling you directly that the configuration setting for two models are incompatible. Oftentimes, the Diagnostic Viewer is much less telling and you might spend an hour or more researching and debugging the error only to end up finding out that what caused the issue was a simple difference between configuration settings for the two models. This is why it is a good paradigm to either develop all models with the same configuration parameters from the get-go or to at the very least create a version of the MasterSim that replaces a link to an external model with the model itself.

“Error in Model block 'HIL_6DOF_MasterSim4_APF_RelOD_TA/NotocodeSubsystem/Prox-1 Hardware and Spacecraft Plant/Prox-1 Hardware Models/Detumble_Controller_Model': the 'Application lifespan' must match between the parent model 'HIL_6DOF_MasterSim4_APF_RelOD_TA' and the referenced model 'TorqueRod_Hardware'. The parent model has a value of '1.0', while the referenced model has a value of 'Inf'. To change this parameter, go to the 'Optimization' page of the Configuration Parameters dialog.”

This is another example of an error caused by dealing with externally-linked models. With this error luckily the Diagnostic Viewer tells you how to fix the problem.

“The extrinsic function 'frame2im' is not available for standalone code generation. It must be eliminated for stand-alone code to be generated. It could not be eliminated because its outputs appear to influence the calling function. Fix this error by not using 'frame2im' or by ensuring that its outputs are unused.”

This is an error caused by your model containing an extrinsic function. In MATLAB, an extrinsic function is one which requires MATLAB to run. Since the function requires MATLAB’s overhead it is not available for standalone code generation. Since the entire point of autocoding is to reduce the overhead of a model without affecting its behavior you should eliminate every output-affecting extrinsic function from the MasterSim and replace it with an autocodeable variant.

This can be more difficult that it appears at first. While some extrinsic functions are relatively simple and can be avoided by adding slightly more long-winded code, some extrinsic functions (especially ones related to image processing) may take days to work around. In these cases, you should read up on as much of the documentation for the extrinsic function as possible and make sure you fully understand how the function works. Then you have the choice of either:

a) Writing a MATLAB function that mimics the extrinsic function. This may be the easier option but it can result in slower code.
b) Writing C code that mimics the extrinsic function and either manually inserting this code into already generated code or importing the C function into MATLAB as a .s file and referencing this function instead of the original extrinsic one.

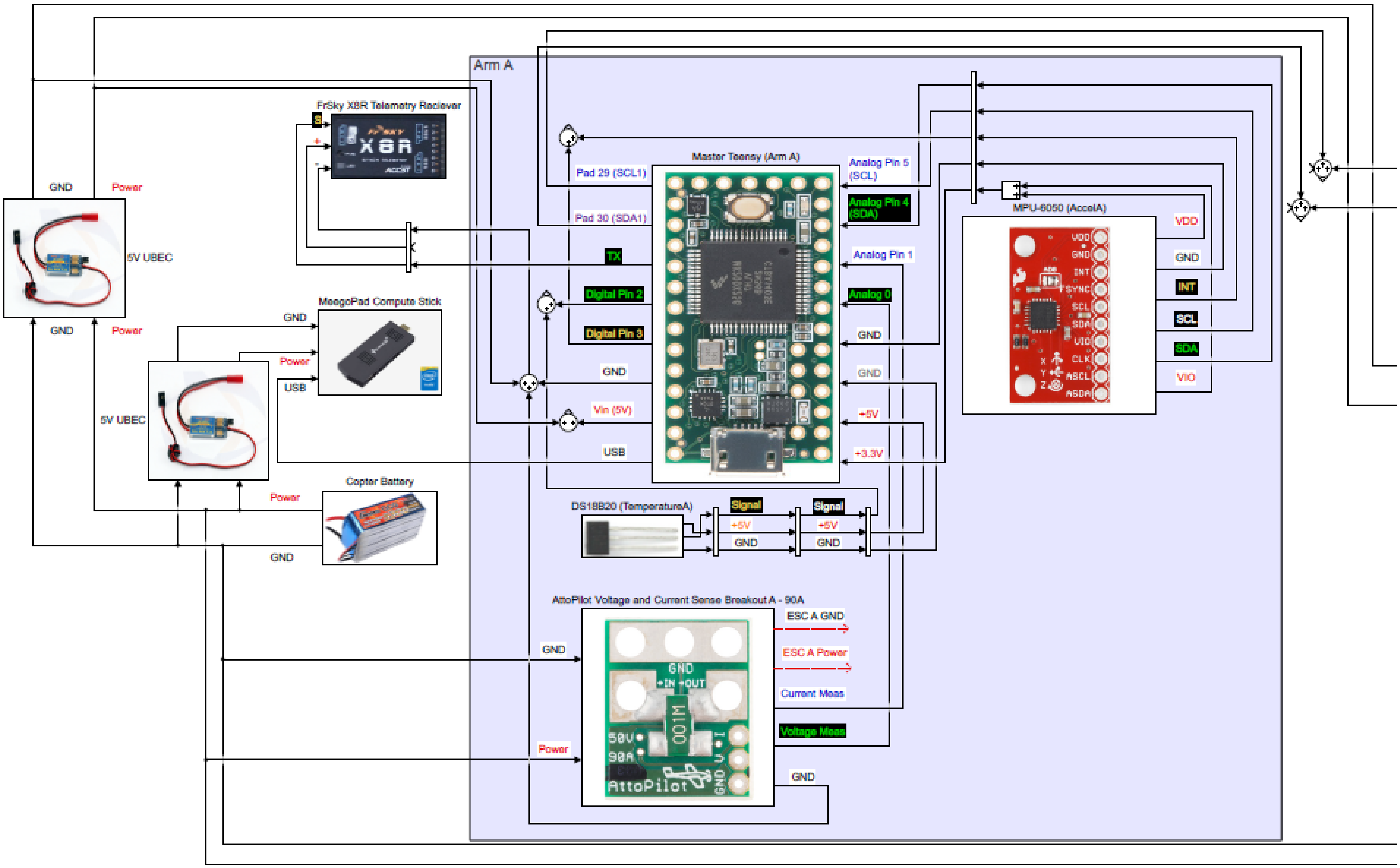
Whichever way you choose, note that you should rigorously test your self-designed function in order to make sure it is equal to Mathworks' version in every conceivable scenario the function would be put through.

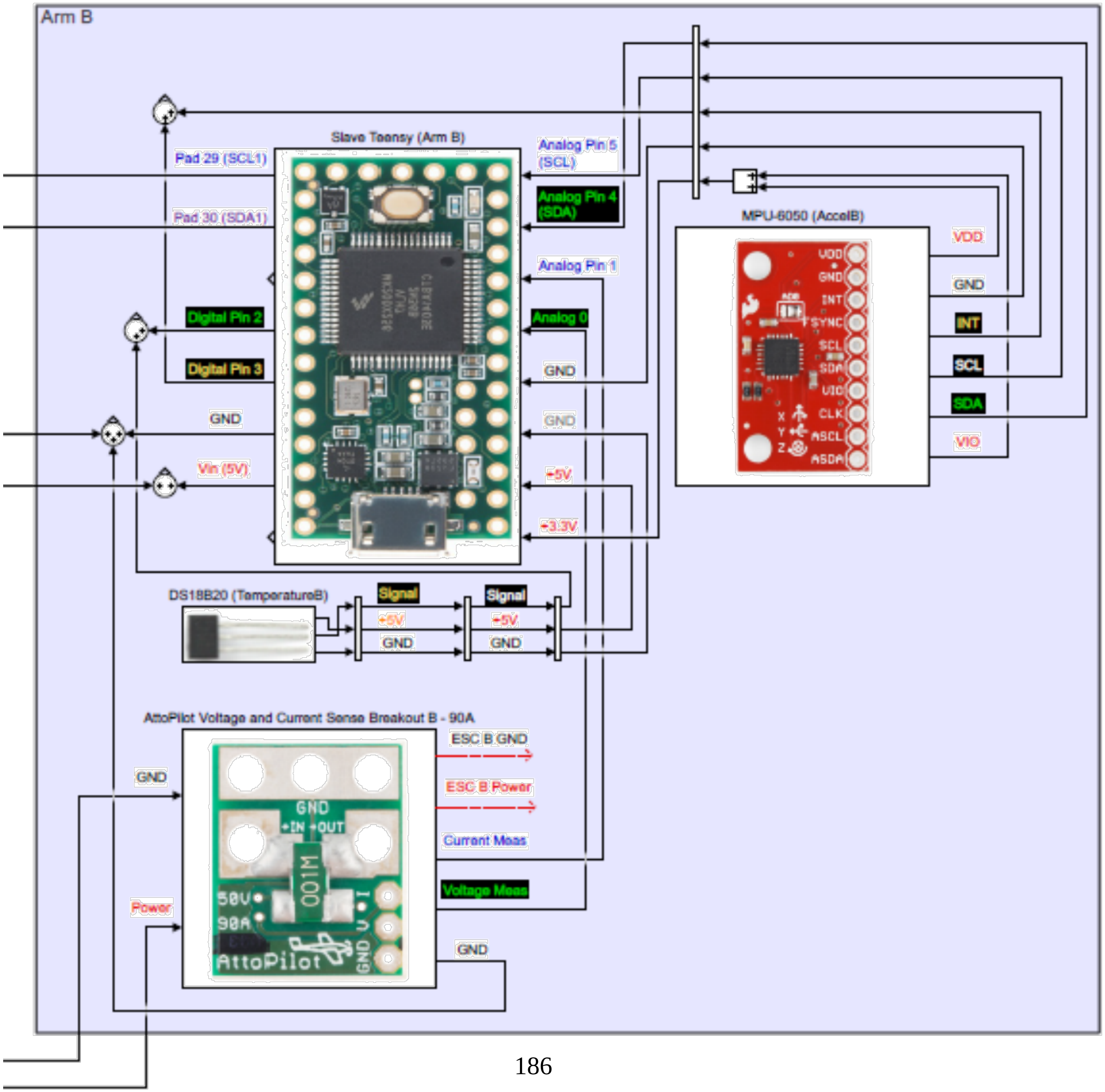
Final Note

Just because your final C code was produced without throwing any errors in MATLAB does not necessarily mean your code is error-free. It would be wise to test the C code independently using the process defined in Section 2 to determine if in fact your model behaves as expected.

APPENDIX B: UAV NERVOUS SYSTEM CONNECTION DIAGRAM

This appendix is referenced in Section 3.6 and shows the electrical connections for the upgraded UAV Nervous System. This version of the nervous system contains two suites of sensors for two separate arms of a rotary wing UAV. Each arm of the copter has a Teensy microprocessor (using Arduino code), an MPU-6050 accelerometer, a DS18B20 temperature sensor, and a 90A AttoPilot voltage and current sense breakout board. One Teensy is powered by a 5V Universal Battery Elimination Circuit (UBEC) connection to the copter's lithium polymer battery, and the second Teensy is powered by a USB connection to the MeegoPad compute stick, which is also powered by a 5V UBEC connection to the lithium polymer battery. Note that only one Teensy processor is connected to the MeegoPad via USB, and sensor data is collected by the other Teensy and passed to the first Teensy over an I2C connection.


Arm A
FrSky X8R Telemetry Reciever
S
+
-
GND
Power
5V UBEC
GND
Power
MeegoPad Compute Stick
GND
Power
USB
5V UBEC
Copter Battery
Power
GND
Master Teensy (Arm A)
Pad 29 (SCL1)
Pad 30 (SDA1)
TX
Digital Pin 2
Digital Pin 3
GND
Vin (5V)
USB
Analog Pin 5 (SCL)
Analog Pin 4 (SDA)
Analog Pin 1
Analog 0
GND
GND
+5V
+3.3V
MPU-6050 (AccelA)
VDD
GND
INT
SCL
SDA
VIO
DS18B20 (TemperatureA)
Signal
+5V
GND
Signal
+5V
GND
AttoPilot Voltage and Current Sense Breakout A - 90A
ESC A GND
ESC A Power
Current Meas
Voltage Meas
GND
GND
Power


Arm B
Slave Teensy (Arm B)
Pad 29 (SCL1)
Pad 30 (SDA1)
Digital Pin 2
Digital Pin 3
GND
Vin (5V)
Analog Pin 5 (SCL)
Analog Pin 4 (SDA)
Analog Pin 1
Analog 0
GND
GND
+5V
+3.3V
MPU-6050 (AccelB)
VDD
GND
INT
SCL
SDA
VIO
VDD
GND
INT
FSYNC
SCL
SDA
VIO
CLK
ASCL
ASDA
X
Y
Z
DS18B20 (TemperatureB)
Signal
+5V
GND
Signal
+5V
GND
AttoPilot Voltage and Current Sense Breakout B - 90A
ESC B GND
ESC B Power
Current Meas
Voltage Meas
GND
GND
Power
GND
+IN
+OUT
001M
50V
90A
I
V
GND
AttoPilot

APPENDIX C: MARS SAMPLE RETURN SUBSYSTEM TAXONOMY

This appendix is referenced in Section 4.5.1 and contains the complete subsystem taxonomy for the Sample Return Orbiter (SRO). It is shown in text format for readability. This includes components for the Rendezvous OS Capture, Telecom, Flight Software, Command & Data Handling, Guidance, Navigation, & Control, Propulsion, Electrical Power, and Thermal Control subsystems. Note that the SRO is still in the preliminary design phase, so these components are listed in a generic and functional way because specific components have not been selected for the mission.

- Ground Support Systems
- Mars Ascent Vehicle (MAV)
- Orbiting Sample container (OS)
- Space-Based Support Systems
- Sample Return Orbiter (SRO)
    - Telecom
        - Antennas & Gimbals
        - Radio Hardware/Software
        - Capture Door Closure Fault
    - Flight Software
        - Fault Protection software
        - VML Sequencing
    - Command & Data Handling
        - Flight Processor
        - Solid State Recorder

  - Data Busses
- Propulsion
  - Reaction Control System (RCS) Thrusters
  - Solar Electric Propulsion (SEP) Thrusters
  - Propellant Tanks
- Electrical Power
  - Solar Arrays & Gimbals
  - Batteries
  - Power Distribution & Busses
- Thermal Control
  - Thermal Sensors
  - Heaters
- Guidance, Navigation, & Control (GN&C)
  - GN&C Mode/Decision Logic
  - GN&C Software Components
    - Attitude GN&C Software
      - Attitude Determination Algorithms
      - Attitude Guidance Algorithms
        - OS Mosaicing Algorithm
        - Attitude Constraints
      - Attitude Control Algorithms
    - Translational GN&C Software
      - Orbit Determination Algorithms

Image Processing Algorithms

Orbit Determination Filters

Orbit Control Algorithms

Orbit Guidance Algorithms

Solar Electric Propulsion Inertial Guidance

Rendezvous Relative Guidance

GN&C Hardware Components

Attitude Determination Sensors

Inertial Measurement Unit

Star Tracker

Sun Sensors

Attitude Control Actuators

Reaction Wheels

RCS Thrusters (propulsion)

Rendezvous Sensors

Narrow Angle Camera (NAC) [hawk]

Medium Angle Camera (MAC) [dog]

Wide Angle Camera (WAC) [fish]

Long-Wave Infrared (LWIR)

LIDAR

Rendezvous OS Capture System (ROCS)

OS Sensors

Laser Curtain

- OS Confirmation Sensor
- Force/Torque Sensors

Flashlight

Capture Volume (container for capturing OS)

Capture Door

Reorientation Hardware

Break-the-Chain Hardware

Earth Entry Vehicle (EEV)

APPENDIX D: FULL FAULT TREE FOR MARS SAMPLE RETURN TERMINAL RENDEZVOUS AND CAPTURE PHASE

This appendix is referenced in Section 4.5.1 and lists the complete GN&C fault tree for the Mars Sample Return autonomous terminal rendezvous and capture phase. A text format is used here for the fault tree rather than a graphical format to improve readability. Faults in bold were selected to be examined in greater detail, as described in Appendix E.

- Failure to Capture the OS
  - Fault During Approach
    - Relative Orbit Determination Fault
      - Rendezvous Sensor Data Fault
        - Sensor Hardware Fault
          - Sensor Loses Power
          - Sensor Settings Incorrect
          - Solid-State Recorder Malfunction
        - Sensor Background Noise
          - Radiation-Induced Noise
          - Temperature-Induced Noise
          - **Stray Light Glint**
        - OS Passes Too Quickly Through Imager FOV
          - SRO Angular Rates Too Great
          - OS Relative Velocity Too Great
        - Sensor FOV Impaired
          - Lens Fogged Due to Outgassing
          - Spacecraft Component in FOV

Debris in FOV

Poor Conditions for OS Tracking

OS Surface Properties Unfavorable

OS Blends Into Background

OS in Eclipse or Shadow (visual only)

Phase Angle Unfavorable

Flashlight Malfunction (visual only)

LIDAR/IR Sensor Faults

No OS Data from Sensors

Orbit Determination Computation Fault

Image Processing Fault

Navigation Software Fault

Mosaicing Algorithm Misses OS

Orbit Perturbations Differ from Models

OS Outgassing Perturbs Orbit

SRO Plume Impingement on OS

Atmospheric Drag Perturbs Relative Orbit

Other Orbit Perturbation Mismodeling

Incorrect Model Parameters (i.e. OS optical properties)

Ephemeris or Timing Fault

Filter Does Not Converge

Guidance & Control Fault

Attitude Fault

Degraded Attitude Knowledge

Inertial Measurement Unit Fault

No Data Output

Reset/Excessive Reset

Bias/Scale Factor Offset

Measurement Drift

Star Tracker Fault

No Output

Bias/Incorrect Output

Excessive Current Draw

Optics Contamination

Optics Coating Degradation

False/Intermittent Star ID

Temporary Lock-Up

Noisy Measurements

Sun Sensor Fault

Attitude Filter Does Not Converge

Degraded Attitude Control

Reaction Wheel Fault

Wheel Stuck/Seized/Not Rotating

Increased Drag/Friction

Excessive Current Draw

Excessive Vibration

Tachometer Fault

Drive Electronics Fault

Wheel Momentum Saturated

Reaction Control System Thruster Fault

Thruster Fails to Actuate

Thruster Stuck On

Tank heater fault

Propellant line freezing

Trajectory Fault

Maneuver Fault

Incorrect Timing, Direction, or Delta-V for Burn

Deadband Violation Does Not Trigger Manuever

Degraded Translational Control

Unable to place OS inside capture cone

OS not aligned with capture cone

Rotation Rate Too High

Relative Velocity Too High

Guidance & Control Software Fault

Sequencing Fault

Spacecraft Reboot

Unable to Meet Conditions that Allow Transfer to a Key State

Tolerances on parameters too tight

Unexpected configuration

Telemetry Reporting Fault

Logical Error in Sequence

Premature Entry into any State

Logical Error in Sequence

Sequencing Software Coding Fault

Ground Command Halts or Unloads Sequence

Incorrect Config File Version Loaded

Fault During Capture

Capture Door Closure Fault

Door Close Timing Fault (Early/Late Closure)

Door Close Signal Does Not Activate

Door Mechanism Fault

Unexpected OS Dynamics

OS Spin Rate Exceeds Capture Requirement

OS Energy Exceeds Capture Capability

OS Impacts ROCS Components

Capture Detection Sensor Fault

Door Sensor Fault

OS Confirmation Sensor Fault

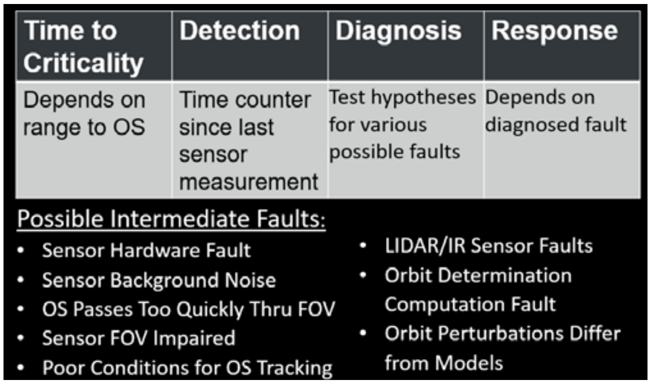
Force/Torque Sensor Fault

Sun Interference/Spoofing

APPENDIX E: PRELIMINARY MSR FAULT PROTECTION STRATEGIES AND REQUIREMENTS

This appendix is referenced in Section 4.5.2 and lists preliminary fault protection strategies and requirements for the selected faults from the Mars Sample Return fault tree in Appendix D. The selected faults are difficult to detect, diagnose, or respond to in a safe and timely manner and have high consequences for autonomous rendezvous and capture.

- Relative Orbit Determination Faults
    - No OS Data From Sensors



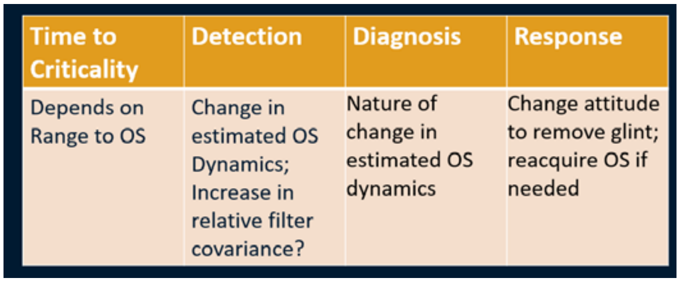
Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Time counter since last sensor measurement	Test hypotheses for various possible faults	Depends on diagnosed fault

Possible Intermediate Faults:

- Sensor Hardware Fault
- Sensor Background Noise
- OS Passes Too Quickly Thru FOV
- Sensor FOV Impaired
- Poor Conditions for OS Tracking
- LIDAR/IR Sensor Faults
- Orbit Determination Computation Fault
- Orbit Perturbations Differ from Models

        - *The flight system shall stop maneuvers if no OS data is received from the rendezvous sensors during the passive abort region (Zone 1) of autonomous rendezvous.*
        - *The flight system shall abort from autonomous rendezvous if no OS data is received from the rendezvous sensors during the active abort region (Zone 2).*
        - *The flight system shall restore measurements of OS position within <30 seconds> (TBR) if no OS data is received from the rendezvous sensors during the unavoidable intercept region (Zone 3).*
        - *The flight system shall restore measurements of OS position within <15 minutes> (TBR) if no OS data is received from the rendezvous sensors during all other subphases.*

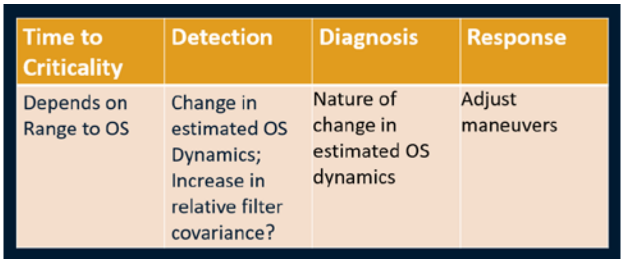
- Stray Light Glint



Time to Criticality	Detection	Diagnosis	Response
Depends on Range to OS	Change in estimated OS Dynamics; Increase in relative filter covariance?	Nature of change in estimated OS dynamics	Change attitude to remove glint; reacquire OS if needed

  - *The flight system shall stop maneuvers if stray light glint on the rendezvous sensors negatively impacts the ability to estimate the OS relative orbit during the passive abort region (Zone 1) of autonomous rendezvous.*
  - *The flight system shall abort from autonomous rendezvous if stray light glint on the rendezvous sensors negatively impacts the ability to estimate the OS relative orbit during the active abort region (Zone 2).*
  - *The flight system shall perform autonomous rendezvous in the presence of stray light glint on the rendezvous sensors during the unavoidable intercept region (Zone 3)*
  - *The flight system shall preserve <40%> (TBR) of nominal relative orbit estimation performance in the presence of stray light glint during all other subphases of autonomous rendezvous.*

- OS Plume Impingement Modeling Fault



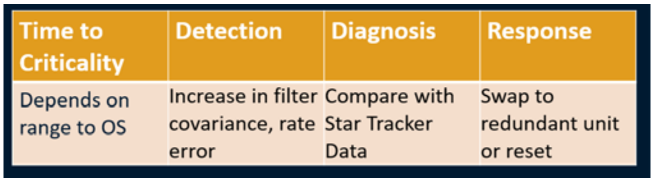
Time to Criticality	Detection	Diagnosis	Response
Depends on Range to OS	Change in estimated OS Dynamics; Increase in relative filter covariance?	Nature of change in estimated OS dynamics	Adjust maneuvers

  - *The flight system shall stop maneuvers if plume impingement causes unmodeled OS dynamics during the passive abort region (Zone 1) of autonomous rendezvous.*
  - *The flight system shall abort from autonomous rendezvous if plume impingement causes unmodeled OS dynamics during the active abort region (Zone 2).*

    - *The flight system shall perform autonomous rendezvous in the presence of unmodeled OS dynamics caused by plume impingement during the unavoidable intercept region (Zone 3).*
    - *The flight system shall perform autonomous rendezvous in the presence of unmodeled OS dynamics caused by plume impingement during all other subphases.*

  - Filter Does Not Converge
    - *The flight system shall stop maneuvers if the relative orbit filter does not converge during the passive abort region (Zone 1) of autonomous rendezvous.*
    - *The flight system shall abort from autonomous rendezvous if the relative orbit filter does not converge during the active abort region (Zone 2).*
    - *The flight system shall restore relative orbit filter estimation within <30 seconds> (TBR) if the relative orbit filter does not converge during the unavoidable intercept region (Zone 3) of autonomous rendezvous.*
    - *The flight system shall restore relative orbit filter estimation within <15 minutes> (TBR) if the filter does not converge during all other subphases of autonomous rendezvous.*

- Attitude Determination & Control Faults
  - Gyro Bias/Scale Factor

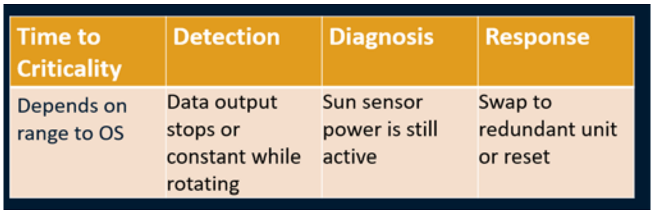


Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Increase in filter covariance, rate error	Compare with Star Tracker Data	Swap to redundant unit or reset

  - *The flight system shall preserve <60%> (TBR) of nominal maneuver performance in the presence of a gyro bias or scale factor during the passive abort region (Zone 1) of autonomous rendezvous.*
  - *The flight system shall perform autonomous rendezvous in the presence of a gyro bias or scale factor during the active abort region (Zone 2).*
  - *The flight system shall perform autonomous rendezvous in the presence of a gyro bias or scale factor during the unavoidable intercept region (Zone 3).*
  - *The flight system shall preserve <40%> (TBR) of nominal maneuver performance in the presence of a gyro bias or scale factor during all other subphases of autonomous rendezvous.*

- *The flight system shall stop maneuvers in the presence of a gyro bias or scale factor during autonomous capture.*

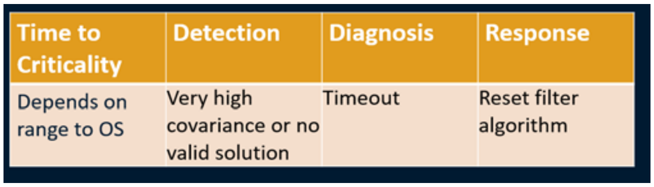
- Star Tracker Temporary Lockup



Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Data output stops or constant while rotating	Sun sensor power is still active	Swap to redundant unit or reset

- *The flight system shall preserve <60%> (TBR) of nominal maneuver performance in the presence of a temporary star tracker lockup during the passive abort region (Zone 1) of autonomous rendezvous.*
- *The flight system shall perform autonomous rendezvous in the presence of a temporary star tracker lockup during the active abort region (Zone 2).*
- *The flight system shall perform autonomous rendezvous in the presence of a temporary star tracker lockup during the unavoidable intercept region (Zone 3).*
- *The flight system shall restore maneuver capability within <15 minutes> (TBR) in the presence of a star tracker temporary lockup during all other subphases of autonomous rendezvous.*
- *The flight system shall stop maneuvers in the presence of a star tracker temporary lockup during autonomous capture.*

- Attitude Filter Does Not Converge

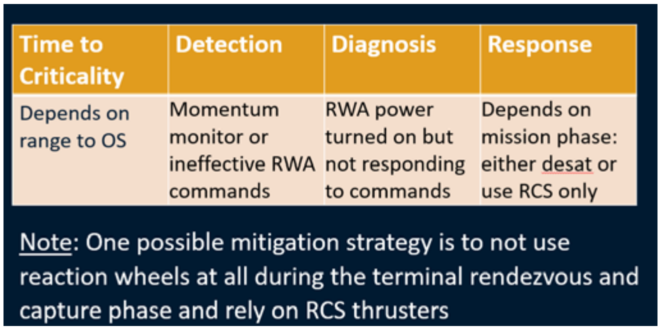


Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Very high covariance or no valid solution	Timeout	Reset filter algorithm

- *The flight system shall stop maneuvers if the attitude filter does not converge during the passive abort region (Zone 1) of autonomous rendezvous.*
- *The flight system shall abort from autonomous rendezvous if the relative orbit filter does not converge during the active abort region (Zone 2).*
- *The flight system shall restore attitude filter estimation within <30 seconds> (TBR) if the attitude filter does not converge during the unavoidable intercept region (Zone 3) of autonomous rendezvous.*

- *The flight system shall restore attitude filter estimation within <15 minutes> (TBR) if the attitude filter does not converge during all other subphases of autonomous rendezvous.*
- *The flight system shall stop maneuvers if the attitude filter does not converge during autonomous capture.*

- Reaction Wheel Momentum Saturated

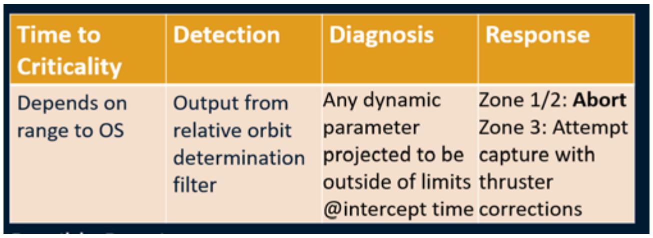


Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Momentum monitor or ineffective RWA commands	RWA power turned on but not responding to commands	Depends on mission phase: either desat or use RCS only

Note: One possible mitigation strategy is to not use reaction wheels at all during the terminal rendezvous and capture phase and rely on RCS thrusters

- *The flight system shall preserve <60%> (TBR) of nominal maneuver performance in the presence of a reaction wheel momentum saturation during the passive abort region (Zone 1) of autonomous rendezvous.*
- *The flight system shall perform autonomous rendezvous in the presence of a reaction wheel momentum saturation during the active abort region (Zone 2).*
- *The flight system shall perform autonomous rendezvous in the presence of a reaction wheel momentum saturation during the unavoidable intercept region (Zone 3).*
- *The flight system shall restore maneuver capability within <15 minutes> (TBR) in the presence of a reaction wheel momentum saturation during standby in a holding location in autonomous rendezvous.*
- *The flight system shall preserve <40%> (TBR) of nominal maneuver performance in the presence of a reaction wheel momentum saturation during all other subphases of autonomous rendezvous.*
- *The flight system shall perform autonomous capture in the presence of a reaction wheel momentum saturation.*

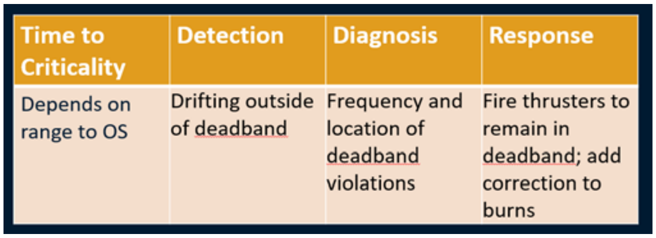
- Trajectory Maneuver Faults
    - OS Dynamics Exceed Capture Requirements



Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Output from relative orbit determination filter	Any dynamic parameter projected to be outside of limits @intercept time	Zone 1/2: **Abort** Zone 3: Attempt capture with thruster corrections

        - *The flight system shall perform autonomous rendezvous even if the projected (to time of capture) OS spin rate exceeds 3 RPM, relative velocity exceeds 5 cm/s, or lateral offset exceeds 10 cm during the unavoidable intercept region (Zone 3).*
        - *The flight system shall abort from autonomous rendezvous if the projected (to time of capture) OS spin rate exceeds 3 RPM, relative velocity exceeds 5 cm/s, or lateral offset exceeds 10 cm during all other subphases.*

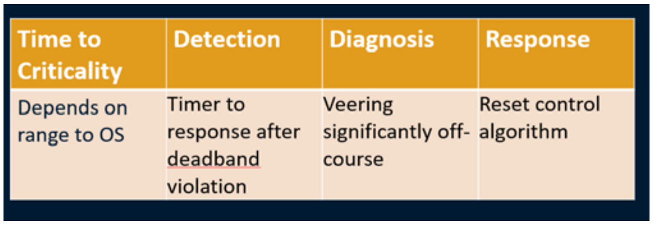
    - Incorrect Timing/Direction/Delta-V



Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Drifting outside of deadband	Frequency and location of deadband violations	Fire thrusters to remain in deadband; add correction to burns

        - *The flight system shall preserve <60%> (TBR) of nominal trajectory performance in the presence of a trajectory maneuver error during the passive abort region (Zone 1) of autonomous rendezvous.*
        - *The flight system shall perform autonomous rendezvous in the presence of a trajectory maneuver error during the active abort region (Zone 2).*
        - *The flight system shall perform autonomous in the presence of a trajectory maneuver error during the unavoidable intercept region (Zone 3).*
        - *The flight system shall preserve <40%> (TBR) of nominal trajectory performance in the presence of a trajectory maneuver error during all other subphases of autonomous rendezvous.*

- Deadband Violation Does Not Trigger Maneuver

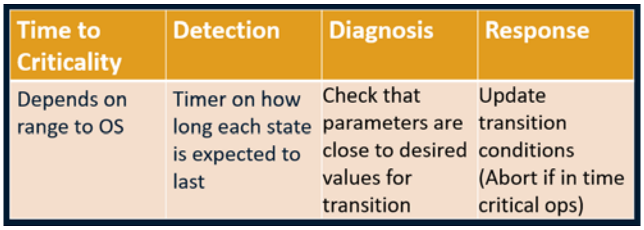


Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Timer to response after deadband violation	Veering significantly off-course	Reset control algorithm

  - *The flight system shall preserve <60%> (TBR) of nominal trajectory performance in the presence of a deadband violation error during the passive abort region (Zone 1) of autonomous rendezvous.*
  - *The flight system shall perform autonomous rendezvous in the presence of a deadband violation error during the active abort region (Zone 2).*
  - *The flight system shall perform autonomous rendezvous in the presence of a deadband violation error during the unavoidable intercept region (Zone 3).*
  - *The flight system shall preserve <40%> (TBR) of nominal trajectory performance in the presence of a deadband violation error during all other subphases of autonomous rendezvous.*

◉ Sequencing Faults

- Tolerances on Transition Parameters Too Tight

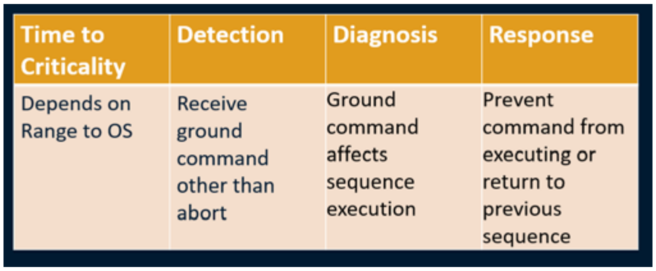


Time to Criticality	Detection	Diagnosis	Response
Depends on range to OS	Timer on how long each state is expected to last	Check that parameters are close to desired values for transition	Update transition conditions (Abort if in time critical ops)

  - *The flight system shall restore nominal sequencing within <10 minutes> (TBR) in the presence of a sequence parameter tolerance error during the passive abort region (Zone 1) of autonomous rendezvous.*
  - *The flight system shall abort from autonomous rendezvous in the presence of a sequence parameter tolerance error during the active abort region (Zone 2).*
  - *The flight system shall perform autonomous rendezvous in the presence of a sequence parameter tolerance error during the unavoidable intercept region (Zone 3).*

- *The flight system shall perform autonomous capture after the OS enters the capture volume in the presence of a sequence parameter tolerance error.*

- Ground Command Halts or Unloads Sequence

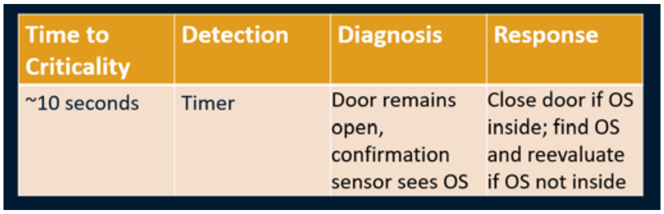


Time to Criticality	Detection	Diagnosis	Response
Depends on Range to OS	Receive ground command other than abort	Ground command affects sequence execution	Prevent command from executing or return to previous sequence

- *The flight system shall restore nominal sequencing within <10 minutes> (TBR) in the presence of a sequencing ground command error during the passive abort region (Zone 1) of autonomous rendezvous.*
- *The flight system shall abort from autonomous rendezvous in the presence of a sequencing ground command error during the active abort region (Zone 2).*
- *The flight system shall perform autonomous rendezvous in the presence of a sequencing ground command error during the unavoidable intercept region (Zone 3).*
- *The flight system shall restore nominal sequencing within <15 minutes> (TBR) in the presence of a sequencing ground command error during all other subphases of autonomous rendezvous.*
- *The flight system shall perform autonomous capture in the presence of a sequencing ground command error.*

- Capture Faults

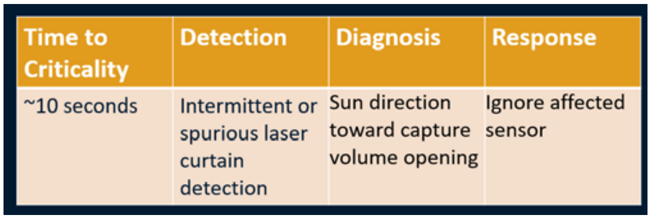
- Door Close Signal Does Not Activate



Time to Criticality	Detection	Diagnosis	Response
~10 seconds	Timer	Door remains open, confirmation sensor sees OS	Close door if OS inside; find OS and reevaluate if OS not inside

- *The flight system shall restore capture door close capability within <10 seconds> (TBR) in the presence of a capture door close signal error during autonomous capture.*

- Sun Interference/Spoofing of Laser Curtain



Time to Criticality	Detection	Diagnosis	Response
~10 seconds	Intermittent or spurious laser curtain detection	Sun direction toward capture volume opening	Ignore affected sensor

  - *The flight system shall maintain <60%> (TBR) of nominal laser curtain sensor detection in the presence of sun interference or spoofing during autonomous capture.*

REFERENCES

[1] N. Dennehy, J.R. Carpenter, NESC Review of Demonstration of Autonomous Rendezvous Technology (DART) Mission Mishap Investigation Board Review (MIB), NASA Engineering and Safety Center Report, RP-06-119, Dec. 2006, http://www.nasa.gov/pdf/167813main_RP-06-119_ 05-020-E_DART_Report_ Final_Dec_27.pdf, (accessed 10/9/15).

[2] L. Fesq, N. Dennehy, et al., Fault Management Handbook, NASA Technical Handbook, NASA-HDBK-1002, Apr. 2012, https://www.nasa.gov/pdf/636372 main_NASA-HDBK-1002_Draft.pdf, (accessed 12/12/16).

[3] J. Winn, C. Bishop, Model-Based Machine Learning, Online Book, http://www.mbmlbook.com/LearningSkills_Learning_the_guess_probabilities.html, (accessed 12/15/16).

[4] A. Zolghadri, The Challenge of Advanced Model-Based FDIR Techniques for Aerospace Systems: The 2011 Situation, Progress in Flight Dynamics, Guidance, Navigation, Control, Fault Detection, and Avionics, Vol. 6, Dec. 2013, pp. 231-248, doi: 10.1051/eucass/201306231.

[5] R.D. Rasmussen, Thinking Outside the Box to Reduce Complexity in NASA Flight Software, NASA Study on Flight Software Complexity, Jet Propulsion Laboratory, Pasadena, California, Mar. 2009, https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf (accessed 10/26/17).

[6] G.J. Cancro et al., Emphasizing Understandability, Flexibility, and Verifiability in a Spacecraft Fault Management Autonomy System, AIAA Infotech@Aerospace Conference, Seattle, WA, Apr. 2009.

[7] G.J. Cancro, APL Spacecraft Autonomy: Then, Now, and Tomorrow, Johns Hopkins APL Technical Digest, Vol. 29, No. 3, 2010, pp. 226-233.

[8] G.H. Horvath, G. Jones, R. Joshi, A Model-Based Approach to Verification of Spacecraft Software using the SPIN Model Checker, AIAA Space 2009 Conference & Exposition, Pasadena, California, Sept. 2009.

[9] A. Wander, R. Förstner, Innovative Fault Detection, Isolation, and Recovery Strategies On-Board Spacecraft: State of the Art and Research Challenges, Deutscher Luft- und Raumfahrtkongress 2012, Berlin, Germany, Sept. 2012.

[10] M. Nakamra, Y. Kawakatsu, C. Hirose, et al., Return to Venus of the Japanese Climate Orbiter AKATSUKI, Acta Astronautica, Vol. 93, 2014, pp. 384-389, doi: 10.1016/j.actaastro.2013.07.027.

[11] J. Day, A. Murray, P. Meakin, Toward a Model-Based Approach to Flight System Fault Protection, IEEE International Conference for Aerospace, Big Sky, MT, Mar. 2012.

[12] J.F. Castet, M. Bareh, J. Nunes, et al., Fault Management Ontology and Modeling Patterns, AIAA Space 2016, Long Beach, California, Sept. 2016

[13] M.D. Ingham, R.D. Rasmussen, M.B. Bennett, and A.C. Moncada, Engineering Complex Embedded Systems with State Analysis and the Mission Data System, Journal of Aerospace Computing, Information, and Communication, 2 (Dec. 2005).

[14] Stateflow Documentation, The Mathworks, Inc., http://www.mathworks.com/help/stateflow/index.html, (accessed 7/6/16).

[15] P.Z. Schulte, D.A. Spencer, Development of an Integrated Spacecraft Guidance, Navigation, & Control Subsystem for Automated Proximity Operations, Acta Astronautica, 118 (Jan-Feb 2016), 168-186, doi:10.1016/j.actaastro.2015.10.010.

[16] N.F. Rouquette, T. Neilson, G. Chen, The 13th Technology of Deep Space One, IEEE Aerospace Conference, Aspen, CO, Mar. 1999.

[17] P.J. Pingree, et. al., Validation of Mission Critical Software Design and Implementation Using Model Checking, IEEE Digital Avionics Systems Conference, Oct. 2002.

[18] M.C. Jackson, J.R. Henry, Orion GN&C Model Based Development: Experience and Lessons Learned, AIAA-2012-5036, AIAA Guidance, Navigation, and Control Conference, Minneapolis, Minnesota, August 2012.

[19] G. Watney, L. Reder, S. Chang, JPL Statechart Autocoder (SCA) Rev. 2 Repository, NASA Jet Propulsion Laboratory, Nov. 2016, https://github.com/JPLOpenSource/SCA. (accessed 12/18/17).

[20] M. Aguilar, Fault Management Using Model Based System Engineering (MBSE) Tools and Techniques, NASA Spacecraft Fault Management Workshop, Sept. 2011,

http://www.nasa.gov/pdf/637605main_day_1-michael_aguilar.pdf. (accessed 10/8/15).

[21] A.M. Homar, AOCS Fault Detection, Isolation and Recovery: A Model-Based Dynamic Verification and Validation Approach, Master's Thesis, Department of Computer Science, Electrical and Space Engineering, Lulea University of Technology, Sept. 2014.

[22] J.F. Castet, M.L Rozek, M.D. Ingham, et al., Ontology and Modeling Patterns for State-Based Behavior Representation, AIAA Infotech @ Aerospace, AIAA SciTech Forum, Kissimmee, Florida, Jan 2015.

[23] B. Van Besien, Investigating Model-Based Autonomy for Solar Probe Plus, Johns Hopkins Applied Physics Laboratory, Dec 2013, http://flightsoftware.jhuapl.edu/files/2013/talks/FSW-13-TALKS/BVB-FSW-Presentation.pdf, (accessed 12/13/16).

[24] N. Muscettola, P. P. Nayak, B. Pell, B. C. Williams, Remote Agent: to boldly go where no AI system has gone before, Artificial Intelligence, Vol. 103, no. 1–2, 1998, pp. 5–47.

[25] J. Marzat, H. Piet-Lahanier, F. Damongeot, E. Walter, Model-based fault diagnosis for aerospace systems: a survey, Proceedings of the Institution of Mechanical Engineers, Part G: Journal of Aerospace Engineering, Jan 2012.

[26] M.B. Brown, S.A. Johnson, An Overview of the Fault Protection Design for the Attitude Control Subsystem of the Cassini Spacecraft, American Control Conference, Philadelphia, Pennsylvania, June 1998.

[27] P.C. Meakin, Cassini Attitude Control Fault Protection Design: Launch to End of Prime Mission Performance, AIAA Guidance, Navigation, & Control Conference and Exhibit, Honolulu, Hawaii, Aug. 2008.

[28] L.M. Fesq, MARPLE: An Autonomous Diagnostician for Isolating System Hardware Failures, PhD Dissertation, Department of Computer Science, University of California Los Angeles, 1993.

[29] K.O. Kolcio, Model-Based Fault Detection and Isolation System for Increased Autonomy, AIAA Space 2016, Long Beach, California, Sept. 2016.

[30] G.A. Johnson-Roth, Mission Assurance Guidelines for A-D Mission Risk Classes, The Aerospace Corporation, Report No. TOR-2011 (8591)-21, June 2011, http://aerospace.wpengine.netdna-cdn.com/wp-content/uploads/2015/04/TOR-20118591-21-Mission-Assurance-Guidelines-for-A-D-Mission-Risk-Classes.pdf, (accessed 12/15/16).

[31] D.A. Spencer, S.B. Chait, P.Z. Schulte, K.J. Okseniuk, M. Veto, Prox-1 University-Class Mission to Demonstrate Automated Proximity Operations, Journal of Spacecraft and Rockets, Vol. 53, No. 5, July 2016, 847-863, doi:10.2514/1.A33526.

[32] D.A Spencer, R. Deshmukh, S. Pujari, G. Guecha, Mars Telecommunications CubeSat Constellation Relay, Georgia Institute of Technology, Dec 2015, http://www.slideshare.net/RohanDeshmukh10/mars-cubesat-telecom-relay-constellationjpl-final, (accessed 12/15/16).

[33] S.B. Chait, D.A Spencer, Georgia Tech Small Satellite Real-Time Hardware-in-the-Loop Simulation Environment: SoftSim6D, Master's Project Report, Georgia Institute of Technology, Dec 2015.

[34] 2015 NASA Technology Roadmaps - TA 4: Robotics and Autonomous Systems, National Aeronautics and Space Administration, July 2015, http://www.nasa.gov/sites/default/files/atoms/files/2015_nasa_technology_roadmaps_ta_4_robotics_and_autonomous_systems_final.pdf, (accessed 10/9/15).

[35] N.W. Green, A.R. Hoffman, H.B. Garrett, Anomaly Trends for Long-Life Robotic Spacecraft, Journal of Spacecraft and Rockets, Vol. 43, No. 1 (Jan-Feb 2006), 218-224, doi: 10.2514/1.14497.

[36] Aerospace Mission Failure Analysis for NASA Ames Research Center Design for Safety Initiative, The Aerospace Corporation, Sept. 30, 2001, http://science.ksc.nasa.gov/shuttle/nexgen/Nexgen_Downloads/ROCKET_FAILURES_FailureCauses_Peinemann.pdf, (accessed 12/13/16).

[37] J.H. Hayes, Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project, 14th International Symposium on Software Reliability Engineering, Denver, CO, Nov. 2003.

[38] B.E. Goldberg, K. Everhart, et al., Systems Engineering "Toolbox" for Design-Oriented Engineers, NASA Reference Publication 1358, Dec. 1994, http://www.hq.nasa.gov/office/codeq/doctree/rp1358.pdf, (accessed 12/13/16).

[39] R.J. Simmons, Fault Tree Analysis, Tunghai University, Feb 2009, http://www2.nuu.edu.tw/~er/reportfile/saminar/Fault_Tree_Analysis.pdf, (accessed 12/13/16).

[40] H.J Kim, W. E. Wong, et. al, Bridging the Gap Between Fault Trees and UML State Machine Diagrams for Safety Analysis, 2010 Asia Pacific Software Engineering Conference, Sydney, Australia, Dec. 2010, 196-205, doi 10.1109/APSEC.2010.31

[41] P.Z. Schulte, D.A. Spencer, N.G. Smith, M.F. McCabe, Development of a Fault Protection Architecture Based Upon State Machines, 67th International Astronautical Congress, Guadalajara, Mexico, Sept. 2016, IAC-16-D1.IP.2x32540.

[42] H. Liu, D. Derawi, J. Kim, Y. Zhong, Robust Optimal Attitude Control of Multirotors, Australasian Conference on Robotics and Automation, Sydney, Australia, Dec. 2013.

[43] SparkFun Triple Axis Accelerometer and Gyro Breakout - MPU-6050, SparkFun Electronics, https://www.sparkfun.com/products/11028, (accessed 7/6/16).

[44] Teensy 3.2, SparkFun Electronics, https://www.sparkfun.com/products/13736, (accessed 7/6/16).

[45] MeegoPad T02 Second Generation Intel Windows TV Stick, http://www.x86pad.com/t02.html, (accessed 7/6/16).

[46] Teensy Arduino Shield Adapter, SparkFun Electronics, https://www.sparkfun.com/products/13288, (accessed 7/6/16).

[47] Predict k-nearest neighbor classification – MATLAB, The Mathworks, Inc., http://www.mathworks.com/help/stats/classificationknn.predict.html, (accessed 7/6/16).

[48] X8R-products, FrSky Electronic Co., Ltd., http://www.frsky-rc.com/product/pro.php?pro_id=105, (accessed 7/6/16).

[49] Taranis X9D Plus, FrSky Electronic Co., Ltd., http://www.frsky-rc.com/product/pro.php?pro_id=137, (accessed 7/6/16).

[50] One Wire Digital Temperature Sensor - DS18B20, SparkFun Electronics, https://www.sparkfun.com/products/245, (accessed 7/6/16).

[51] AttoPilot Voltage and Current Sense Breakout - 90A, SparkFun Electronics, https://www.sparkfun.com/products/9028, (accessed 12/18/17).

[52] Gens ace 5300mAh 22.2V 30C 6S1P Lipo Battery Pack – Gens Ace, Genspow GmbH, http://www.gensace.de/gens-ace-5300mah-22-2v-30c-6s1p-lipo-battery-pack.html, (accessed 12/18/17).

[53] P.Z. Schulte, D.A. Spencer, State Machine Fault Protection for Automated Proximity Operations, 68th International Astronautical Congress, Adelaide, Australia, Sept. 2017, IAC-17-C1.5.11x36573.

[54] P.Z. Schulte, D.A. Spencer, Fault Protection for Mars Sample Return Autonomous Rendezvous & Capture, Symposium on Space Innovations, Atlanta, Georgia, Oct. 2017.

[55] S. Ueda, T. Kasai, H. Uematsu, HTV Rendezvous Technique And GN&C Design Evaluation Based on 1st Flight On-orbit Operation Result, AIAA/AAS Astrodynamics Specialist Conference, Toronto, Canada, Aug 2010.

[56] E. De Pasquale, ATV Jules Verne: a Step by Step Approach for In-Orbit Demonstration of New Rendezvous Technologies, 12th International Conference on Space Operations, Stockholm, Sweden, June 2012.

[57] R.B. Friend, Orbital Express Program Summary and Mission Overview, Sensors and Systems for Space Applications II Conference, Orlando, Florida, Mar 2008.

[58] M. Delpech, J.C. Berges, S. Djalal, et. al., Preliminary Results of the Vision Based Rendezvous and Formation Flying Experiments Performed During the PRISMA Extended Mission, IAA-AAS-DyCoSS1-12-07, Advances in the Astronautical Sciences, Vol. 145, 2012, pp.1375-1390.

[59] M. Sabatini, M., G.B. Palmerini, P. Gasbarri, A Testbed for Visual Based Navigation and Control During Space Rendezvous Operations, 65th International Astronautical Congress, Toronto, Canada, Oct 2014.

[60] D.A. Spencer, Automated Trajectory Control for Proximity Operations Using Relative Orbital Elements, PhD Dissertation, Georgia Institute of Technology, May 2015.

[61] S. Nolet, Development of a Guidance, Navigation, and Control Architecture and Validation Process Enabling Autonomous Docking to a Tumbling Satellite, Ph.D.

Dissertation, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 2007.

[62] V.H. Nguyen, A.K. Chaudhary, D. Poladian, et al., The GN&C of the SMV (Space Maneuver Vehicle) Flight Test Vehicle: Rapid Design of an Unpowered Autolanding System, AAS GN&C 98-024, Annual AAS Rocky Mountain Guidance and Control Conference, Breckenridge, Colorado, February 1998, pp. 264-265.

[63] R. Da Costa, M. Markus, G. Ortega, Rapid prototyping tool for development and validation of GN&C onboard software, 54th International Astronautical Congress, Bremen, Germany, Vol. 3, October 2003, pp. 1287-1294.

[64] S.M. Stewart, L. Ward, S. Strand, Distributed GN&C Flight Software Simulation for Spacecraft Cluster Flight, AAS 14-032, 37th Annual AAS Guidance and Control Conference, Breckenridge, Colorado, Jan-Feb 2014.

[65] Vision and Voyages for Planetary Science in the Decade 2013-2022, Committee on the Planetary Science Decadal Survey, National Research Council of the National Academies, 2011, https://solarsystem.nasa.gov/docs/131171.pdf (accessed 9/4/17).

[66] R. Mattingly, L. May, Mars Sample Return as a Campaign, 2011 Institute of Electrical and Electronics Engineers Aerospace Conference, Big Sky, Montana, Mar. 2011. doi: 10.1109/AERO.2011.5747287.

[67] J.E. Riedel, J. Guinn, et al., A Combined Open-Loop and Autonomous Search and Rendezvous Navigation System for the CNES/NASA Mars Premier Orbiter Mission, 26th Annual AAS Guidance and Control Conference, Breckenridge, Colorado, USA, Feb 2003.

[68] D. Henry, C. Le Peuveic, L. Strippoli, F. Ankersen, Model-based Fault Detection Isolation and Recovery and Fault Accommodations for a Rendezvous Mission around the Mars Planet: the Mars Sample Return Case, 4th International Federation of Automatic Control International Conference on Intelligent Control and Automation Sciences 2016, Reims, France, Vol. 49, No. 5, June 2016. doi: 10.1016/j.ifacol.2016.07.124.

[69] T. Ormston, Time delay between Mars and Earth, Mars Express Blog, European Space Agency, http://blogs.esa.int/mex/2012/08/05/time-delay-between-mars-and-earth/, (accessed 4/11/18).

[70] B.V. Semenov, SPICE Reference Frames, NASA Ancillary Information Facility, https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/C/req/frames.html, (accessed 4/2/18).

[71] T.A. Lovell, D.A. Spencer, Relative Orbital Elements Formulation Based upon the Clohessy-Whiltshire Equations, Journal of the Astronautical Sciences, Vol. 61, No. 4, Feb. 2015, pp. 341-366. doi: 10.1007/s40295-014-0029-6.

[72] C.A. Grasso, VML 3.0 Reactive Rendezvous and Docking Sequencer for Mars Sample Return, AIAA SpaceOps Conference, Pasadena, California, USA, May 2014.

[73] L. Walker, "Automated Proximity Operations Using Image-Based Relative Navigation" 26th Annual USU/AIAA Conference on Small Satellites, Logan, Utah, August 2012, SSC12-VII-3.

[74] P.Z. Schulte, D.A. Spencer, M. Goggin, Mars Sample Return Terminal Rendezvous Fault Protection, Journal of Spacecraft and Rockets, submitted Apr. 2018.

[75] P.Z. Schulte, D.A. Spencer, Generic State Machine Fault Protection Architecture for Aerospace Vehicle Guidance, Navigation, & Control, Journal of Aerospace Information Systems, submitted Jun. 2018.

[76] P.Z. Schulte, D.A. Spencer, Development of an Integrated Spacecraft Guidance, Navigation, & Control Subsystem for Automated Proximity Operations, 65th International Astronautical Congress, Toronto, Canada, Oct. 2014, IAC-14-C1.6.4x21108.

[77] K.J. Okseniuk, S.B. Chait, P.Z. Schulte, D.A. Spencer, Prox-1: Automated Proximity Operations on an ESPA Class Platform, 29th AIAA/USU Conference on Small Satellites, Logan, Utah, Aug. 2015.

[78] P.Z. Schulte, D.A. Spencer, On-Board Model-Based Fault Diagnosis for Autonomous Proximity Operations, 69th International Astronautical Congress, Bremen, Germany, Sept. 2018, IAC-18-C1.6x45016.

[79] P.Z. Schulte, J.W. Moore, A.L Morris, Verification and Validation of Requirements on the CEV Parachute Assembly System Using Design of Experiments, AIAA-2011-2558, 21st AIAA Aerodynamic Decelerator Systems Conference and Seminar, Dublin, Ireland, May 2011.

[80] P.Z. Schulte, E.G. Lightsey, K.B. Brumbaugh, R.L Staehle, Utilization of a Solar Sail to Perform a Lunar CubeSat Science Mission, 2nd Interplanetary CubeSat Workshop, Ithaca, New York, May 2013.

[81] Pellegrino, M., Gibson, A., Mariscal, J.C., Schulte, P., "UNISPACE+50: Shared Vision, Common Action," 68th International Astronautical Congress, Adelaide, Australia, Sept. 2017, IAC-17-E3.1.1x37185.