View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

Transitioning Autonomous Systems Technology Research to a Flight Software Environment

Jeremy Frank
Gordon B. Aaseng[1]
NASA Ames Research Center, Moffett Field, CA 94035-1000

NASA has developed methods and algorithms for autonomous spacecraft operations, including automated planning and scheduling, fault diagnostics and impact determination, procedure management and display. Making the transition from technology research to operational flight software requires overcoming significant technical, programmatic and cultural challenges. Technology research is aimed at developing methods that perform specific functions correctly, but the resulting software may not be designed for flight processors with limited CPU, memory and network resources, and may not be easily integrated into spacecraft flight software. Our objective in the Autonomous Systems and Operations Project is to make significant strides toward the transformation from technology to operational use. Our focus was twofold: maturing research grade autonomy software into a flight software environment using broadly accepted languages and tools; and integrating autonomy applications with each other and with representative systems and their data and command interfaces. For a target flight software environment, we chose Core Flight Software, developed by Goddard Space Flight Center as a common operating system independent framework. Our hardware integration environment was provided by the Integrated Power and Avionics Systems (iPAS) Lab at Johnson Space Center, in which various subsystem development has been conducted to address engineering challenges for the vehicles and systems required for long-duration missions into the solar system. The iPAS and its network of connected facilities provides realistic subsystem hardware or simulations of spacecraft power, life support, guidance, navigation and control, and command and data handling subsystems. Interfaces between autonomy applications and the subsystems being assessed and controlled were developed, assessed and refined. The hardware and software environment using CFS and the iPAS facility has proven to be a highly flexible and realistic environment in which to rapidly integrate applications in an iterative, low cost setting. Using the integration environment we have developed, we will turn our focus to performance and sizing analysis to determine the computational requirements for full-scale deployment of autonomy technology. Scalability of reasoners and the spacecraft models upon which they operate, and robustness across the full range of spacecraft conditions and environments will be explored and improved. We are making significant contributions to the future programs that will build the spacecraft that will take humans beyond the Earth-Moon system, in which program Systems Engineers will be able to accurately and confidently design in accurate, robust and mature autonomous operations systems.

Nomenclature

ACAWS = Advanced Caution and Warning System
AMPS = Advanced Modular Power System
APC = Autonomous Power Controller
ARC = Ames Research Center
ASO = Autonomous Systems and Operations
BSP = Board Support Package
CDD = Command and Data Dictionary
cFS = Core Flight Software

[1] Computer Scientist, Intelligent Systems Division, MS 269-1.

cFE = Core Flight Executive
CH_4 = Methane Molecule
CO_2 = Carbon Dioxide Molecule
cRIO = Compact Real-Time Input/Output
DS = Data Store
ECLSS = Environmental Control and Life Support System
EFT-1 = Exploration Flight Test #1 (Orion Spacecraft)
EPS = Electrical Power System
GRC = Glenn Research Center
H_2 = Hydrogen Molecule
HyDE = Hybrid Diagnostic Engine
IMS = Inductive Monitoring
iPAS = Integrated Power and Avionics System
JSC = Johnson Space Center
ISS = International Space Station
LADEE = Lunar Atmosphere and Dust Environment Explorer
LRO = Lunar Reconnaissance Orbiter
MBSU = Modular Bus Switching Unit
MCC = Mission Control Center
MID = Message Identifier
NASA = National Aeronautics and Space Administration
O_2 = Oxygen Molecule
OS = Operating System
OSAL = Operating System Abstraction Layer
PDU = Power Distribution Unit
PLB = Programmable Load Bank
PLEXIL = Plan Execution Interchange Language
PPA = Plasma Pyrolysis Assembly
PPC 750 = Power Personal Computer
PSP = Platform Support Package
QSI = Qualtech Systems, Inc.
RBI = Remote Bus Isolator
RPC = Remote Power Controller
SB = Software Bus
SBC = Single Board Computer
SBN = Software Bus Network
SCM = Software Configuration Management
VM = Virtual Machine

I. Motivation and Objectives

Future human spaceflight exploration missions will require considerably more autonomy than has been used for crewed missions in the past. The two-way light time places an absolute constraint on communications, and any decisions or reactions that must be completed under the two-way light time must be performed autonomously by the in-space segment (the spacecraft itself and the crew). Autonomy is defined as “operating without external support or assistance”. Autonomy can be achieved by any combination of crew actions and vehicle system automation. During the first global circumnavigation, Ferdinand Magellan’s fleet was completely autonomous in that all navigation and operational decisions were made solely by his fleet of ships, although automation was almost non-existent. A Mars or long-distance asteroid mission will require total autonomy for short-term decisions that must be made in less than the two-way light time between the spacecraft and Earth, as well as during planned and unplanned loss of communications. Increased autonomy will also alleviate the degradation of interactive, conversational communications as the light-time increases. Apollo, Shuttle and International Space Station (ISS) have relied predominantly on Earth-based Mission Control Centers (MCCs) for mission decisions.

NASA's Autonomous Systems and Operations Project (ASO) has developed or extended several technologies to automate these higher order decisions or provide the kind of integrated information to flight crews that enable them to make complex decisions without reliance on off-board support. Autonomous operations require both nominal and off-nominal decision-making. Nominal decisions and controls are needed to ensure that activities are performed as planned, and that operational constraints are enforced. However, as noted by Carl Von Clausewitz, "No battle plan ever survives contact with the enemy"; a similar maxim holds for spaceflight. Off-nominal conditions are always unexpected and unplanned; even though the probability of failures, and to some extent the timing of failures using prognostic technology, can be estimated, system failure and degradation almost certainly requires both human and system decision-making, and also induces some perturbation into operational plans and schedules. These can range from minor and recoverable perturbations, to whole-scale mission replanning, such as situations that require an early return or change in mission objectives.

NASA's ASO Project, as well as several other research groups, have been conducting technology research into the decision automation methods that will be required for effective autonomy. These include planning and scheduling systems, procedure automation and assistance, and fault management detection and isolation, anomaly detection, failure prognostics, and failure impact determination. Significant progress has been made over the past several years to develop and mature intelligent software capable of making the types of complex decisions required of an autonomous spacecraft. These technologies have been integrated as a decision aid for astronauts to demonstrate autonomous mission operations, both in analog environments and onboard the ISS[1,2,3,4]. Having substantially accomplished major proof of concept and feasibility demonstrations, the next steps include proving scalability, robustness and performance attributes in realistic flight environments. However, previous demonstrations used automation and, more importantly, flight software technology developed decades ago; while these demonstrations show what is possible, novel flight software and decision support technology will form the foundation of the next generation of human spaceflight missions[4].

In order to become a part of an on-board critical decision and control system, successful autonomy technology must be converted to flight quality and standards, must fit within constrained performance allocations, and must scale up to spacecraft dimensions from the system subsets used by technology projects for concept development. The infusion into programs must eventually be adopted by a flight program, but programs, with good justification, have been reluctant to adopt unproven autonomy technology. Among the justifications for the reluctance are:

- Additional autonomy technology is not required to meet the mission requirements.
- Accurate and trustworthy metrics needed to estimate the cost of a full-scale implementation are not available, resulting in substantial program cost and schedule risk.
- Processor, memory, and bandwidth data for the technology are not available with enough accuracy to make reliable sizing, power and thermal estimates, resulting in significant technical risk.

Maturing the existing technology base to address these concerns requires a significant development effort. Typically, software technology is built on common computing resources such as modern Intel processors, with Linux or Windows Operating Systems, and non-real-time interfaces and networks. Software environments have been chosen to promote rapid development rather than using commonly accepted flight software; software in combinations of Java, Python, C++ and web tools are common[1,3,4]. Furthermore, technology research has not been conducted with system performance and robustness as a primary goal. Finally, software has been built to technology research lab standards to prove feasibility rather than developing for execution in flight-like environments on constrained computing resources. Our current focus is on creating an environment in which to rigorously examine and analyze performance of autonomy applications, using representative systems and their data to extrapolate performance of future exploration spacecraft and missions. On completion of performance analysis and the derivation of usable parametrics to aid in scaling performance to full systems, we will be able to better address autonomy requirements and their implementation costs.

We have selected a set of autonomy applications and converted them to execute on processors and software that is either flight qualified or on a viable path to flight certification, integrated with realistic, representative flight-like subsystems with data, failure spaces and operational decisions similar to current and future human spacecraft systems. We have chosen the Power PC (PPC 750) processor, the VxWorks Operating System and the Core Flight Software (cFS) middleware as the flight-like environment for the next phase of technology research and "infusion research". VxWorks Operating System has been flown on numerous uncrewed missions; cFS, originally developed by Goddard Space Flight Center as a common, reusable operating system independent interface layer[5], has been used recently on the Lunar Atmosphere and Dust Environment Explorer (LADEE), the Lunar Reconnaissance Orbiter (LRO), and other missions. Autonomy applications have been refactored in the C or C++ programming language to be compatible with cFS and flight software standards, and performance improvements have been made where feasible. Even though software is not developed to rigorous software processes and standards, the refactored

applications can be used to derive parametrics that provide accurate estimations of the processor, memory and throughput of applications scaled up to full-sized spacecraft systems, and to pinpoint areas where additional performance optimization is warranted.

In the remainder of this paper, we will describe the autonomy technology applications that we have developed and are converting from technology research to a path to flight. The processing hardware and software environments that we have selected for the next step on the path to flight will be discussed, including the basis for selecting the chosen systems and tools and descriptions of some of their key attributes. Next we will describe the subsystems that we have used to create realistic operational settings in which to execute the autonomy applications, including the types of failure cases and decision reasoning that stress the applications enough to extrapolate from the testbed setting to spacecraft scales. The results of the integration and testing that we have completed will be described, with our lessons learned, and we will conclude by describing the benefits that are anticipated with the applications, environments, and the integration and test environment that has been developed.

II. Autonomy Technology Applications

We have selected a group of autonomy applications and test domains to be executed in the flight processing environment from technologies that have been developed to varying states of maturity over the past several years. In this section we will describe the applications and their purpose, their maturity and heritage, the systems and subsystems for which they have been used in the past as well as on this project. Since the applications, the associated subsystems, and the software and processing environment are closely associated, we provide a brief introduction to the cFS and environment and the subsystems prior to describing the autonomy technology applications.

The cFS software environment provides standard services for control of applications and inter-process communications. A key feature is the Software Bus, which is a logical construct by which applications communicate. The Software Bus can extend across multiple processors, whether they are located adjacently or distributed at long distances. Several common services, including Time Services, Event Services, Execution Services and Table Services provide many of the core functions required of any real-time flight software environment. cFS includes a set of standard, reusable applications that perform functions common to multiple applications, including a process scheduler, a data Limit Checker, a Data Storage application for logging application data, and others.

The subsystems with which the autonomy applications interface range from technology testbeds under development and evaluation by other NASA projects to the Orion spacecraft. The Advanced Modular Power System (AMPS) is developed at Glenn Research Center, and is also deployed at Johnson Space Center for integration with other technology development projects. The Plasma Pyrolysis Assembly (PPA) is part of a technology suite for closed loop Environmental Control and Life Support Systems (ECLSS), that recovers molecular hydrogen (H_2) from methane (CH_4) that is produced as a byproduct of oxygen (O_2) recovery from carbon dioxide (CO_2)[6]. The PPA is developed at Marshall Space Flight Center. A Lithium Ion battery, similar in design to the Orion spacecraft battery, is used in conjunction with fault and anomaly detection applications. Initial testing was done with representative data emulating the battery sensors, but recent updates will enable direct connections of the autonomy application software to the battery hardware available in our integration lab. We also have the Orion Exploration Flight Test 1 (EFT-1) data available that will be used for a larger scale performance test in the next year.

The autonomy applications represent a group of autonomy types that we have defined that constitute a range of activities in a decision process, from determining the state of the system to deciding what to do in response to a perturbation of the system. The autonomy application classes are:

- State Estimation – abstract spacecraft data to higher-order descriptions.
- Fault Diagnosis – identification of the root cause faults.
- Failure Impacts Determination - consequences of faults on mission capability, for example, a power system fault results in loss of rendezvous sensor, therefore ability to complete rendezvous is compromised.
- Planning – determining what actions to perform.
- Plan execution – sending commands based on the plan.

The autonomy applications have been demonstrated in settings ranging from technology testbeds to on-board the ISS. The applications, their autonomy classes, their purpose and most mature use prior to our current work, are summarized in Table 1. The Advanced Caution and Warning System (ACAWS)[7] consists of three separate applications to perform different fault management functions, while other applications can operate independently or integrated with other applications.

Table 1 – The autonomy applications, their classes, a brief description and usage prior to the transition to flight environment that is the focus of our current work are described

Autonomy Application	Autonomy Class	Purpose	Prior Usage
ACAWS Fault Detector	State Estimation	Identifies indications of faults from system data, e.g. threshold exceedance	Ground demo with Orion EFT-1
ACAWS Diagnostic Engine	Fault Diagnosis	Combines results of Fault Detector to identify component faults	Ground demo with Orion EFT-1
ACAWS Failure Impacts Reasoner	Failure Impacts Determination	Identifies the components whose functions have been impacted by one or more system faults	Ground demo with Orion EFT-1
Hybrid Diagnosis Engine (HyDE)	State Estimation, Fault Diagnosis	Uses a state-based model with system data to identify failures in terms of transitions to off-nominal system states	On-board ISS demonstration with life support autonomy demonstration
Inductive Monitoring System (IMS)	State Estimation	Anomaly detection system; “trained” with nominal data to recognize anomalies as deviations from known nominal	Ground use for ISS mission support; Ground demo with Orion EFT-1; On-board ISS demonstration with life support autonomy demonstration

Our current work does not include automated planning applications. Planning applications tend to be more complex and their transition to flight environments warrant more dedicated and focused work than our current project is able to perform. Other projects have begun to implement planning functions using the Plan Execution Interchange Language (PLEXIL) in other flight-like environments that can be integrated with our autonomy development at a later time. The plan execution function in the APC is developed independently at Glenn Research Center as part of the AMPS system described in detail in later sections. It executes on a separate processor that is not executing the cFS environment, but is integrated to the extent that data is exchanged with our applications to demonstrate the use of failure information applications with plan execution systems.

The system and software architecture including the cFS software environment, our autonomy applications, and the subsystems used to exercise and test the autonomy, are shown in Figure 1. The autonomy applications, shown in magenta circles, communicate with each other and with the external subsystems via cFS Software Bus messages and use cFS services for execution. Reusable cFS applications support our applications, including the Limit Checker that is used to identify faulted conditions, and Data Storage that provides run-time logging of data. AMPS, PPA and Orion data is connected to the autonomy applications via the Software Bus.

We can now describe the autonomy applications in more detail, with the maturation toward flight-readiness that has been achieved and discuss remaining challenges. Each of the applications in Figure 1 are described in the following sections.

ACAWS Fault Detector. The Fault Detector is the State Estimation element of ACAWS. It obtains data from the target system, performs filtering, cleaning and framing, then conducts tests on the data, and sends the results to the ACAWS Diagnostic Engine as a vector of PASS, FAIL or UNKNOWN test results. Since telemetry data is often imperfect, the Fault Detector first checks the data for any status information available from the target system and conducts additional quality testing as needed. Off-scale or otherwise physically improbable data (30 °K, for example), or data that has uncharacteristically stopped updating for an extended time, is excluded from testing. Once the data has been filtered to exclude as much of the noisy data as possible, tests can be conducted on the data. Tests can be as simple as a red-line or threshold test, or arbitrarily complex algorithms. The Fault Detector can be configured with a persistence setting on each test result to further filter transient results, such as needing three failure results in a row before asserting a FAIL result to the Diagnostic Engine. The Fault Detector also includes a Test Suppression feature that can suppress any test result under certain conditions. For example, if a switch is turned off by the operators, any tests on the load that is turned off can be set to UNKNOWN in order to prevent a misdiagnosis of a failure when it is not in use. The Fault Detector uses the cFS Limit Checker application for testing the AMPS data for conditions symptomatic of system faults. The Limit Checker uses a table to describe the data value to be tested, and a threshold value or condition that indicates a FAIL condition.

ACAWS Diagnostic Engine. The test results and system configuration data from the Fault Detector are sent to a Diagnostic Engine application that includes a commercially available diagnostic reasoner, TEAMS-RT®, to perform

the Fault Diagnosis autonomy function. TEAMS-RT[8] is a library module developed by Qualtech Systems, Inc. (QSI) that contains the run-time artifacts derived from a system fault model built with the companion tool from QSI, TEAMS Designer. The model operates on a matrix that associates the tests with failure modes for a given system configuration to determine the root cause failure that best explains the vector of PASS, FAIL and UNKNOWN results. The diagnostic results are returned from TEAMS-RT API calls and are packaged into a cFS Diagnostic Results message that is transmitted to other applications.

ACAWS Failure Impacts Reasoner. The ACAWS Failure Impacts Reasoner receives diagnostic results from the Diagnostic Engine to perform the Failure Impact Determination autonomy function. The Diagnostic Engine sends a request to the Failure Impacts Reasoner which initiates a query of the system model to determine the components and functions affected by a failure. Failure impacts include both loss of function of a component and loss of redundancy of a required resource to a component. The results can be displayed to notify operators of a comprehensive view of the state of the system following failures – the root cause failure, the components unavailable for use, and components that would be lost by one more failure. The loss of redundancy information is the key to determining next worst failures that help make important decisions about the mission and to prepare for potential additional loss of critical functions.[9]

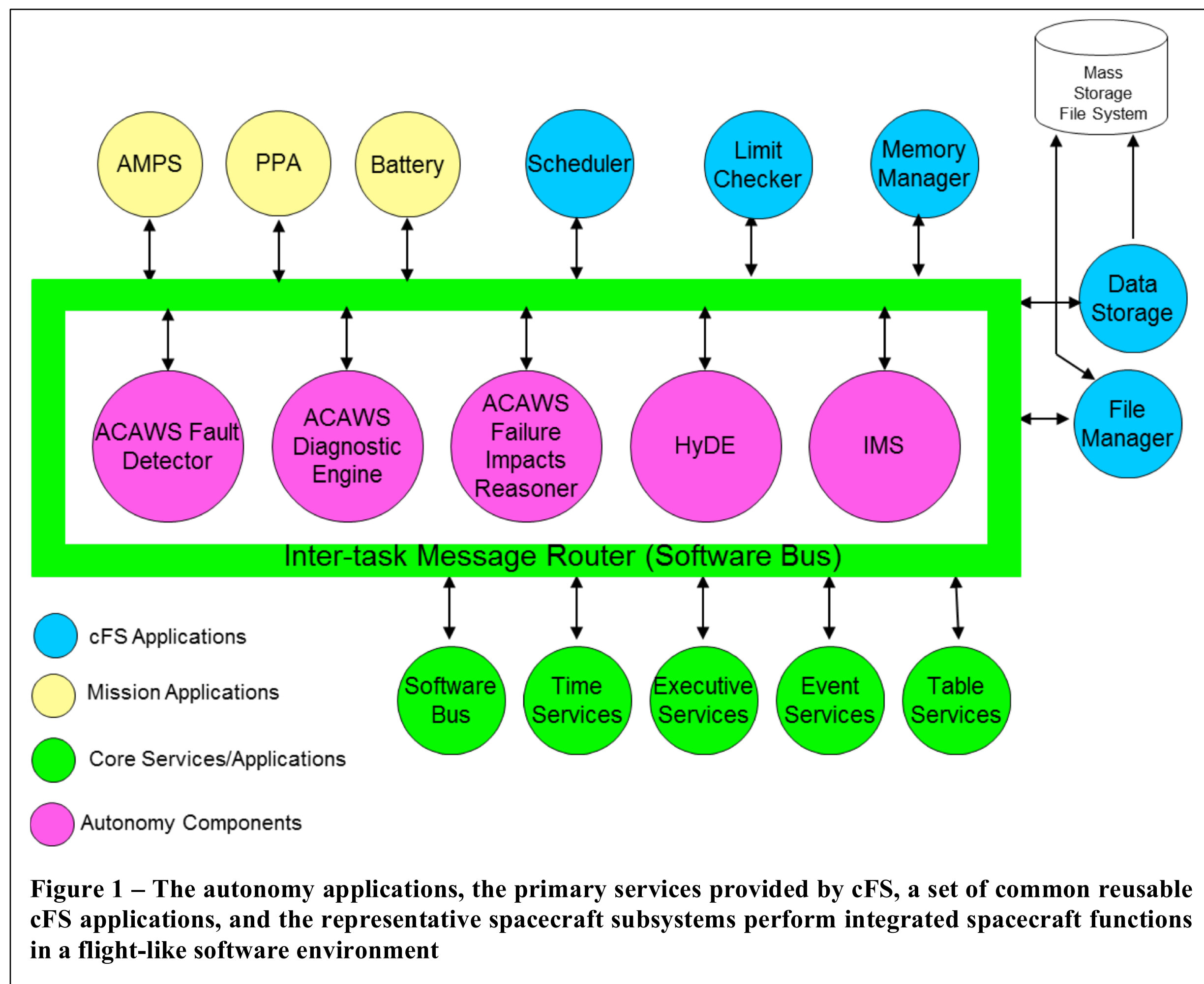

Figure 1 – The autonomy applications, the primary services provided by cFS, a set of common reusable cFS applications, and the representative spacecraft subsystems perform integrated spacecraft functions in a flight-like software environment

HyDE. The Hybrid Diagnostic Engine (HyDE) is a model-based fault diagnosis engine that diagnoses discrete faults. HyDE takes as input a model that describes the combined discrete and continuous behavior of the components of the system[10]. The model consists of discrete modes of operations, relevant variables and parameters of the modes, and how these variables relate to each other in different modes of operations. HyDE uses sensor data from the system and the model of the system behavior to deduce the evolution of the state of the system over time, including changes in state indicative of faults. Transitions between modes are based on either commands or conditions on the internal variables. Faults are represented as special unknown event transitions. HyDE supports

modeling of variables and relations in different domains like Boolean, enumeration, real-valued and interval-valued. HyDE's reasoning algorithm tries to determine the modes of all components, values of all variables, and the occurrence of one or more unknown events in the system. The reasoning starts with the assumption that no unknown events have occurred, and all modes and variables have their initial values. Any available sensor observations are compared against predictions from the model. Inconsistencies between predicted and observed states generates conflicts; each conflict is a set of unknown event transitions that contribute to each inconsistency. A search over the space of unknown events driven by these conflicts results in the generation of one or more fault states that would resolve the inconsistencies. The number of possible faults is configurable, which limits the possible explanations. Further simulation with these potential candidates would determine which potential candidates become actual diagnoses. This process continues over time as more and more observations become available.

IMS. The Inductive Monitoring System (IMS) application is an empirical anomaly detection system to perform a State Estimation function[11]. Instead of seeking a diagnosis of a well-characterized potential failure, the IMS looks for unusual, unexpected system behavior. Using data from various configurations and environments, the IMS is "trained" with known nominal behavior. During operations, IMS detects anomalies as deviations from the normal data. This is usually performed by measuring the 'distance' between the system's current behavior and the characterization of 'nominal' behavior; if this distance exceeds a tunable threshold, it is considered 'off-nominal'. An anomaly may be an unusual but acceptable condition, in which case the training can be updated to include the unusual condition, or it may indicate a malfunctioning or degrading system. The output is a scored value that can alert an operator of unusual conditions that warrant further investigation. Since IMS has been exercised and validated with several systems, the primary objective was to execute it in the cFS environment in order to prepare for performance analysis, rather than to emphasize detection of anomalies.

Autonomous Power Controller. An Autonomous Power Controller (APC) developed at Glenn Research Center (GRC) demonstrates a power system specific Planning and Execution function[12]. It receives diagnostic messages from the ACAWS Diagnostic Engine that is used to initiate automated failure responses, but it also modifies the system state model that the APC uses for any other automated decisions, such as future environmental and system configuration changes. The APC is an early capability autonomous controller, demonstrating the concept of integrating multiple types of reasoners to achieve automated, autonomous controls of a spacecraft subsystem. The APC developers plan to transition the controller to a cFS flight environment as the system matures; our current work with the APC conceptually demonstrates the Plan Execution function that will be matured with more capable planners and execution functions.

III. Subsystem Domains

The autonomy applications described in the previous section are generic and reconfigurable; they are designed to support multiple spacecraft subsystems and interact across subsystem boundaries to properly respond to failures and events that affect multiple subsystems. To effectively test and evaluate the autonomy applications, we selected several spacecraft subsystems that provide realistic problem sets with flight-like sensor and processor data as inputs to the autonomy applications. None of the applications were designed to work specifically with these subsystems; to the contrary, they are all designed to operate with any system. All of the applications require configuration; ACAWS and HyDE require a set of failures and a set of well-defined conditions that map system behavior to a (set of) failures that my have occurred. IMS requires a characterization of the normal performance of a system, which is used to identify off-nominal behavior. We selected a group of subsystems that are themselves in a technology research and maturation stage with engineering testbeds capable of exhibiting realistic nominal and faulted behavior, capable of generating data compatible with cFS software environments, but that are small and agile enough to adapt to the autonomy application test and evaluation needs. The subsystems include Electrical Power Systems (EPS) and Environmental Control and Life Support Systems (ECLSS) components. The subsystems are either physically located in, or have data connectivity to, the Integrated Power and Avionics System (iPAS) testbed at Johnson Space Center. The subsystems and the autonomy applications are shown in Table 2. The subsystem domains provided a good range of problem types that effectively exercised several key attributes of the autonomy applications. Power systems tend to exhibit rapid failure propagation, while ECLSS is frequently characterized by gradual onset such as fluid leaks, and failures of either subsystem can impact the other. Nominal and faulted behavior were simulated where necessary, and actual fault data from testbed execution, when it occurred, was recorded to be played back to the autonomy applications.

Table 2 – The subsystems used for test and evaluation of the autonomy technology applications, with the classes of autonomy functions of each application

Subsystem	Autonomy Application	Reasoning Types
AMPS	ACAWS	State Estimation, Fault Diagnosis, Failure Impacts
Battery Power	IMS	State Estimation
Plasma Pyrolysis Assembly (PPA)	HyDE	State Estimation, Fault Detection
AMPS	APC	Plan Execution

A. Electrical Power Subsystems

The AES Modular Power System (AMPS)[13] was the primary spacecraft system for exercising the ACAWS applications. AMPS, shown in Figure 2, includes several power distribution and power management components with numerous system sensors including current, voltage, switch positions, commanded positions and error conditions. The data is initially processed and converted to engineering units on a Compact Real-time Input Output (cRIO) device, timestamped and transmitted in a CCSDS message format. The sensor data is transmitted via Ethernet to a cFS application that receives the incoming messages and republishes the data in a cFS message format. The combination of the cRIO with a message format converter is an analog of a typical spacecraft's subsystem level processor that performs data collection, packaging and engineering unit conversion, and transmits the data for processing by a supervisory level flight computer.

The AMPS contains power management and distribution hardware, sensors and software representative of space-capable power systems, with redundant components and power channels. Battery power is available from Lithium Ion batteries and emulated solar array power. Modular Bus Switching Units (MBSUs) control which power source is used, and provide power to Power Distribution Units (PDUs). Power redundancy is provided at the bus level with each MBSU sending power to both PDUs, and the PDUs select which MBSU power is used. Sensors include numerous current and voltages, switch positions and error data, such as a mismatch between a commanded state and actual state of switches.

Each MBSU and PDU contains a compact Remote Input Output (cRIO) device that processes data and commands. The cRIO reads sensors, converts signals to engineering units, timestamps the data and transmits it to a central processor. It also receives commands and converts them to electrical signals to control switches. The cRIO uses LabVIEW software package that constructs data messages in a CCSDS format compatible with the cFS messages described in the following sections. The cRIO performs limited error checking, such as comparisons of commanded states and actual measured states of switches, and can report errors such as mismatches. The cRIO is powered by redundant power supplies on each of the inputs to assure power to the cRIO as long as power is available on any of the inputs.

The AMPS system provided a good range of electrical system faults, requiring use of sensor data as well as the loss of data or data validity to identify failures. Electrical faults are characterized by rapid onset and discrete state changes, such as from on to off, or voltage at either 120 volts or 0 volts, whereas faults in the powered equipment are often more continuous with gradual change in temperature, pressure and electrical current consumption. AMPS failures are simulated in hardware with use of commands that are not visible to control and fault management software. Simulated failures include:

- Switch failures.
- Shorts on power cables/connectors.
- MBSU and PDU component-level failures ("dead box").
- Load overcurrent.

Switches are software-commanded Remote Bus Isolators (RBI) in the MBSU, or Remote Power Controllers (RPC) in the PDU. The MBSU RBIs use a switch on both the positive (hot) line and negative (return) line, both commanded to the same position by the cRIO command logic. A fail open condition could occur if either switch opens without command, and sensors report the state of both switches as well as a mismatch between the hot and return line, and mismatches between commanded state and actual state. Likewise, a failed closed condition could occur in which the switch fails to turn off when commanded, or changes from open to closed without command. The dual switch design makes a failed closed fault less likely than a failed open, since two switches would have to inadvertently close, but conditions other than electro-mechanical failure could occur that would close the switch without command.

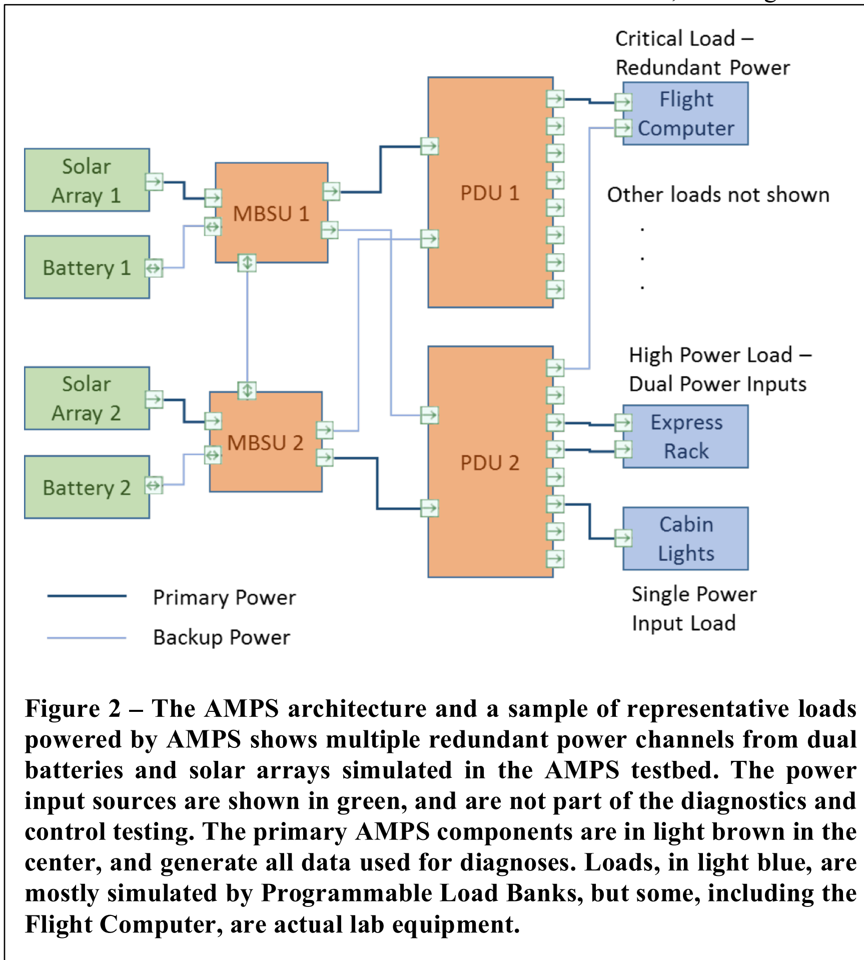

Figure 2 – The AMPS architecture and a sample of representative loads powered by AMPS shows multiple redundant power channels from dual batteries and solar arrays simulated in the AMPS testbed. The power input sources are shown in green, and are not part of the diagnostics and control testing. The primary AMPS components are in light brown in the center, and generate all data used for diagnoses. Loads, in light blue, are mostly simulated by Programmable Load Banks, but some, including the Flight Computer, are actual lab equipment.

Power short conditions result in a trip of the RBI or RPC, and are characterized by a trip without a prior overcurrent condition. The current sensors are processed at 1 Hz so it is quite unlikely that a current spike will be detected to confirm the trip, so it is possible for a “false trip” to occur that would be indistinguishable from a true short circuit. Shorts in the MBSU are simulated by setting a trip limit below the actual current draw. If the current is observed rising toward the trip threshold prior to a trip over at least a few seconds, a short circuit on the power cable would not be diagnosed. A slow increase in current draw is indicative of malfunction of a load, rather than a fault in the wiring.

Component failures of the MBSU or PDU are characterized by a complete loss of both data and power output. Both the MBSU and PDU contain a Compact Real-time Input/Output (cRIO) processor, powered by internal power supplies. The MBSU has a power supply on each of the input power channels, and any of them can power the cRIO, while the PDU has a single power supply. Either a cRIO fault or loss of power to the cRIO would result in identical conditions – a loss of all power and data output - and because all the data from the component requires an operable cRIO, it is not possible to determine if the failure is due to the cRIO or loss of power to the cRIO.

Load faults are simulated by Programmable Load Banks (PLB) that can be programmed to emulate the power profile of typical spacecraft components. The PLB does not simulate the sensors that would be associated with typical equipment, such as temperatures, pressures, or RPMs, so diagnosis of faults is made on the basis of current draw only. Current is monitored, and if it rises above a threshold for at least several seconds, a load overcurrent fault is diagnosed. It would be very likely that current will drop to 0 after an overcurrent fault, either because the current continues to rise and exceeds the trip threshold of the RPC, or because an operator or automated controller responds to the fault (using the diagnostic system’s outputs, perhaps) and commands the overloaded equipment off. Since the fault still exists, the diagnostic system needs to continue to report the overcurrent fault, using a ‘latching’ mechanism that reports the failure until there is positive evidence that the fault was cleared.

The iPAS contains Lithium-Ion batteries that provide power to AMPS system that was used with an anomaly detection system that is trained with nominal data, and uses deviations from nominal to identify anomalies. The Inductive Monitoring System (IMS) was used in the iPAS to identify such battery anomalies. An anomaly is simply

an unusual or unexpected condition; it may turn out to be the result of an atypical but valid environment or configuration, or it could be due to a system malfunction or degradation. Used in combination with other diagnostic systems, it can help identify incipient conditions, or can identify signs of faults that were not known or anticipated. Because the battery system data has only recently been collected, processed and published to cFS, our IMS work has had to rely on simulated, recorded data that contains representative anomalies. Because IMS is relatively mature in its original Linux version and is used in the MCC for ISS anomaly detection, it was not a priority to show that it can detect the battery anomalies; the emphasis was in executing it in cFS and preparing it for performance analysis in a flight environment.

B. ECLSS Subsystems

Failures in Environmental Control and Life Support Systems (ECLSS) are often characterized by more gradual onset and uncertainty. Failures such as fluid leaks tend to occur slowly, with effects that propagate even slower. A gradual pressure drop, perhaps followed in time by temperature increases, could result from loss of coolant fluid; but if operating in an already cold environment, or not too much heat was being generated because systems happened to be operating in quiescent modes, the temperature rise expected of a coolant leak might not occur. Diagnostic systems for these types of failures require different techniques for detecting thresholds, rates of change or more complex determinants of off-nominal system performance.

Our project had access to ECLSS equipment that could be used to exercise diagnostic applications geared to more continuous and uncertain failure behavior. The Plasma Pyrolysis Assembly (PPA) is a component of a system for oxygen (O_2) recovery from carbon dioxide (CO_2). Current oxygen recovery results in methane as a byproduct that is vented overboard, requiring a continuing supply of hydrogen (H_2) as a reactant. The H_2 is obtained from breaking water down into H_2 and O_2, so recovering the H_2 is a major improvement over current CO_2 recovery processes. The PPA recovers the hydrogen from methane and improves the oxygen recovery significantly.

The PPA system controls chemical and physical processes and is sensitive to deviations and tuning. Off-nominal conditions could result from component failures, but can also indicate tuning or balancing of power, temperature control, reagent levels and other parameters required for a well-controlled chemical reactions that convert methane to molecular hydrogen (H_2) and solid carbon. Off-nominal conditions can be detected by conditions such as gradual changes in the reflected microwave power or changes in pressure. These changes may indicate a system failure, may indicate a tuning problem, or may be a normal part of the chemical process, so considering all the data as a state change is ideal for the PPA problem set.

The PPA team provided sample representative data containing several conditions of interest, obtained when the PPA exhibited faults during testing. The PPA faults are detectable primarily with the microwave reflected power and pressure changes. High reflected power indicates that the microwaves are not being absorbed properly in the system, which could be a result of a few possible failures or degradation conditions. The faults present in our PPA datasets included:

- Carbon buildup – a buildup of carbon clogs the system, slowing down or stopping the ability to strip molecular hydrogen from methane.
- High reflected power.
- Sabatier over-pressure, similar to a PPA over-pressure.

A HyDE model was developed for the PPA system to identify several types of failures or off-nominal conditions. The HyDE reasoner uses a system model that correlates state changes with failures by determining if the state change is commanded or a natural result such as changing environmental conditions, and if not, it determines the most likely fault that would cause the observed state change. Like IMS, HyDE is a mature technology, so the primary motivation of our work was to integrate HyDE with cFS and prepare it for performance analysis in a flight like environment. The HyDE reasoner has successfully detected and diagnosed the conditions using the recorded data. The work to date has shown that the diagnostic concept and interfaces to PPA data are correct, clearing the path to a more full-scale integrated capability to detect and diagnose failures and degradation over complex continuous data.

IV. Processor and Software Architectures

The conversion of technology research applications to flight-like applications involved several significant activities that will be described in detail below, including

- Conversion to flight-supported software languages. Applications originally written in Java, Python or other languages were ported to C or C++.

- Implementation in the CFS framework. Applications written for Linux, Windows or other environments were refactored to use cFS services for initialization, events, time management and data handling.
- Standardization of inter-process communications. Technology research interfaces included various combinations of open source, commercial or custom interfaces. These were converted to CFE Software Bus messages in most cases.
- Software optimization. Technology research applications were frequently not built with performance as a primary goal, so as they were ported to C, some applications were significantly refactored to improve processor and memory utilization, particularly in cases of known performance issues from prior technology research.

Software Conversion. For our project, this was primarily porting applications from Java to C, or scripting languages such as Python to C. Such changes can be a large effort that doesn't show substantial differences to outside observers and program managers, but it proved well worth the effort to convert to languages typically used in flight software rather than work with adapters and interfaces that might have allowed use of the original software code. We explored options for retaining legacy software, such as using C to Java interfacing methods such as JNI, but ultimately we determined that continuing to use legacy Java or other software would just delay a necessary transition, and decided to "bite the bullet" and convert the software at the outset. Some software transitioned relatively easily to C, while other applications needed features of C++ to avoid substantial rework. We determined that using C++ libraries invoked by C applications was a viable trade-off. Initial testing using Linux in the CFE framework was very successful with the approach. Subsequent transitions to VxWorks required building C++ support into the VxWorks kernel but otherwise proceeded smoothly.

Implementation in the cFS framework. The cFS environment provides standardized application initialization, registration, execution loops, and shutdown. A small set of cFS services provide time and event management, command and telemetry routing via a Software Bus, and error messaging and handling. Coupled with CFS is an Operating System Abstraction Layer (OSAL) that is intended to achieve application portability between operating systems and processors, and "promote the creation of portable and reusable real time embedded system software. Given the necessary OS abstraction layer implementations, the same embedded software should compile and run on a number of platforms ranging from spacecraft computer systems to desktop PCs."[14] Common functions for queues, semaphores, tasks, timers, file I/O, file system management and interrupts are used by software applications rather than the native operating system functions. Using the OSAL eliminates the need for changing API calls when software moves from Windows to Linux to VxWorks or other operating systems. This also was somewhat tedious work with little initial outward benefit, but clearly pays off when moving an application from early development and test on Linux to executing on embedded processors. Converting applications developed as technology demonstrations to operating system independent applications using the OSAL was a manual process requiring developers to identify the native functions that were included in the OSAL and convert from operating system native functions to the OSAL function. The OSAL is not strictly necessary in order to function or to be flight certified. The recently completed AMO TOCA SSC experiment[4] was flown on the ISS without platform independence provided by the OSAL, but it was executed only on a commodity PC in a virtualization environment, and did not need platform independence.

An example of the benefits of the OSAL can be seen in the file open function. In Linux, the file open API is:

int open(const char **path*, **int** *oflag*, **...);**

with an optional 'int mode' parameter; in VxWorks, the file open function is defined as:

int open(const char *path, int oflag, int mode);

with a required mode flag. The OSAL defines a function

OS_open (const char *path, int32 access, uint32 mode)

that is used instead of the native operating system's 'open' function that will work correctly on any operating system. Applications that are fully "OSAL-compliant" can be readily moved from one operating system to another without code modification because the OSAL handles all the variability between operating systems and their various versions. However, it proved somewhat difficult to be sure that all application developers had truly converted all commonly used functions to the OSAL equivalent. More than once, an application that was considered to be OSAL

compliant failed to execute when moved to another operating system, and software integrators found native operating system function calls still in use were not available or were in a different form on the targeted platform. While not a major problem for the moderate scale of the applications used in the infusion research, this will be an ongoing concern as larger projects make similar transitions. Code reviews and automated tools to check for OSAL compliance could alleviate the concern.

Standardization of Interprocess Communications. Interprocess communications included both communication between independent tasks running on a single processor and between applications running on different processors. The cFS framework includes a Software Bus that transports data between applications on a processor, and an application called Software Bus Network (SBN) facilitates interprocessor communications by extending the Software Bus between processors. All interprocessor communications using SBN relied on Ethernet with standard IP protocols, although conceivably the SBN functionality could be implemented for some other transport protocol. The Software Bus uses a publish and subscribe protocol. The publishing application does not require awareness of what applications may receive the data, and the subscriber does not need to know where the application generating the data is located. The SBN application creates data channels, or ‘pipes’, between all processors that register and transmits messages destined for a subscriber on another processor from the local Software Bus to another SBN node which receives the incoming message and places it on its local Software Bus. Each application uses a set of unique message IDs to correctly identify the data message by the Software Bus routing mechanisms. The Software Bus with the SBN applications effectively extend the Software Bus across all processors in the spacecraft architecture so that applications maintain processor independence with respect to interprocess communication.

While flight programs, such as ISS and the Orion spacecraft have dealt with managing large-scale data communications between processors and various elements of a ground infrastructure, the management of the data has not always been efficient. The cFS framework and the incipient tools we developed to help manage interprocess communications provide opportunity for significant improvements in interprocess data management. Managing the message IDs required careful attention and tools. The architecture allows for multiple copies of an application to execute, but each one needs unique message IDs or its data could be misinterpreted by receiving applications. Indeed, during development and testing, we occasionally encountered conflicts because messages were not properly deconflicted. For smaller scale projects, developers could assign message ID ranges for each application, but as the number of applications, and the number of messages used by the applications, grows, it became clear that message ID automation would be required. A tool called the Collaborative Data Dictionary was developed to help manage the message IDs. The complete specification of the applications, the messages they could generate, and the commands to which they respond remained a complex process.

V. Integration and Testing

The Integrated Power and Avionics System (iPAS) was established at Johnson Space Center as a collaborative environment that has proven to be well-suited for technology infusion activity. The iPAS hosts several subsystem technology testbeds, and also provides secure data communications between NASA centers, providing a range of subsystems with which to interact. Our autonomy applications require realistic nominal and faulted behavior to exercise diagnostics, impact determination and automated plan execution functionality. The iPAS offers both a rich set of subsystems as well as the computational and communications resources in a well-controlled environment in which to test and evaluate the autonomy applications.

We developed a flexible processor architecture to run several applications distributed between processors, and to move applications between processors easily. One of our objectives was to assure that software loads can be balanced between processors either in design, development, or in operations such as a response to loss of computing assets or changing operational needs.

The cFS environment is designed to make the transition between processors seamless. During development and test it is advantageous to begin development and test in an environment such as Linux, move to the target operating system, such as VxWorks, as the design matures, and later transition to the flight hardware when it becomes available. cFS has successfully facilitated these kinds of transitions and we have used its transportability features to address both software distribution and flexibility in our autonomy architecture. Distribution between multiple processors requires establishing data and command interfaces across a network. Flexibility in addition requires that the distribution scheme can be changed with minimal change to software. We successfully showed that flexibility can be achieved with a very small set of configuration data and with no change whatsoever to application software and data.

We executed the set of applications described above in several processor configurations, starting with all applications executing on a single Linux Virtual Machine, migrating to two, then three Linux machines, and finally to a mix of Linux and Power PC processors. Our primary test configuration consisted of the AMPS hardware and APC application, both of which execute outside of the cFS framework, but connected to a Linux cFS node, and ASO applications distributed between a Linux node and PPC with VxWorks, as shown in Figure 3. The dissimilar architecture assures that the architecture and applications properly handle data byte ordering, timing and message integrity.

The following sections will describe the some of the integration and test activities and issues encountered. We will provide details on the message traffic generated by our applications as currently implemented, the work and challenges involved in transitioning cFS applications on Linux to PPC and VxWorks, and several issues that arose during development and integration.

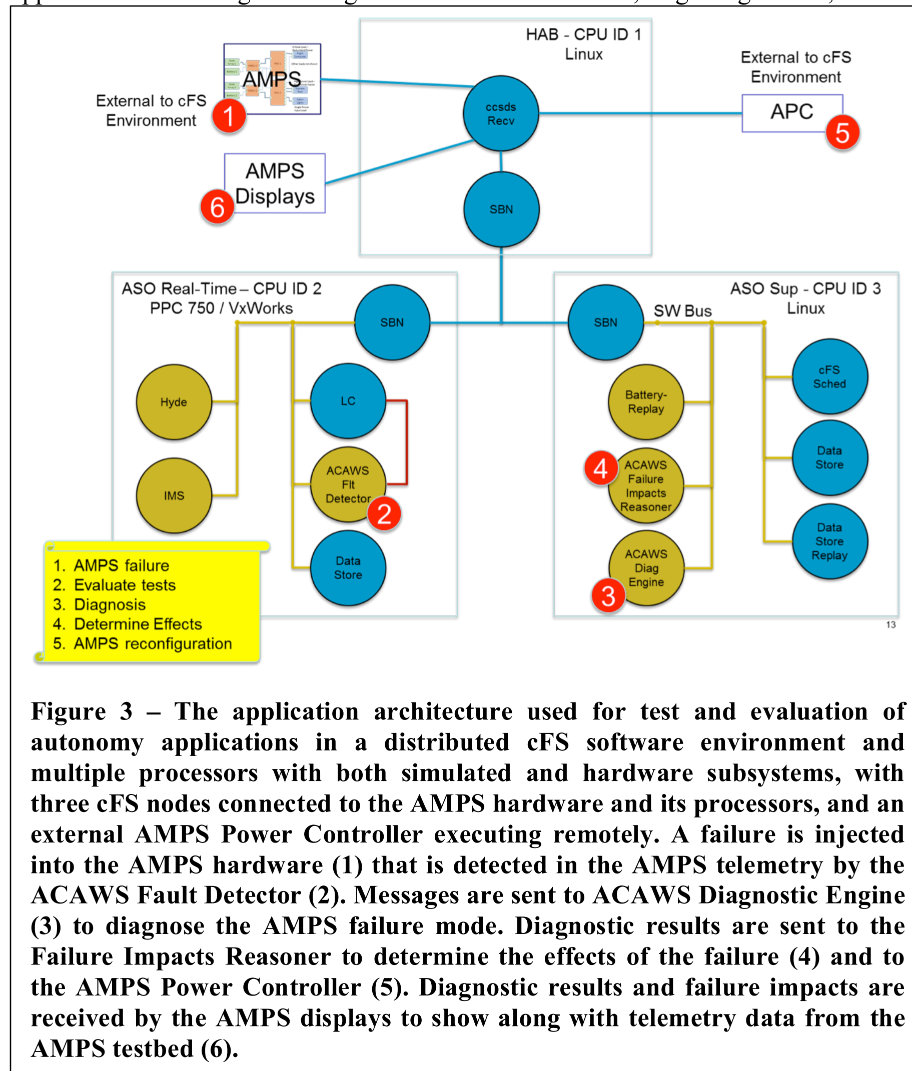

Figure 3 – The application architecture used for test and evaluation of autonomy applications in a distributed cFS software environment and multiple processors with both simulated and hardware subsystems, with three cFS nodes connected to the AMPS hardware and its processors, and an external AMPS Power Controller executing remotely. A failure is injected into the AMPS hardware (1) that is detected in the AMPS telemetry by the ACAWS Fault Detector (2). Messages are sent to ACAWS Diagnostic Engine (3) to diagnose the AMPS failure mode. Diagnostic results are sent to the Failure Impacts Reasoner to determine the effects of the failure (4) and to the AMPS Power Controller (5). Diagnostic results and failure impacts are received by the AMPS displays to show along with telemetry data from the AMPS testbed (6).

A. Message Traffic

The messages between applications all use a fixed size synchronous cyclic data packet. cFS can use asynchronous messaging, but since one of our objectives is to establish a system on which good quality performance metrics can be collected, we determined that cyclic messaging would stress the messaging architecture more, and would be easier to analyze steady state performance without having to determine worst case data bandwidth and the probability of reaching worst case. Our initial data bandwidth has been fairly low, but serves to establish a performance baseline when scaled up applications are exercised later. The message rates and sizes are described in Table 3.

Table 3 – Message descriptions for the inter-process data used in the initial test and evaluation

Message	Rate (Hz)	Size (Bytes)	Generating Application	Receiving Application
ACAWS Test Results	1	73	ACAWS Fault Detector	ACAWS Diagnostic Engine
ACAWS Diagnostic Results	1	140	ACAWS Diagnostic Engine	Autonomous Power Controller, Displays, Data Logs
ACAWS Impact Request	1	38	ACAWS Diagnostic Engine	ACAWS Failure Impact Reasoner

ACAWS Impact Results	1	500	ACAWS Failure Impact Reasoner	Displays, Data Logs
PPA Diagnostic Results	1	133	PPA HyDE Reasoner	Displays
PPA Data	1	264	PPA Data Playback	PPA HyDE Reasoner
IMS Anomaly Scores	1	14	IMS	Displays
AMPS MBSU Data (2 instances)	1	166	AMPS Test Rig	ACAWS Fault Detector
AMPS PDU (2 instances)	1	130	AMPS Test Rig	ACAWS Fault Detector
Total		1,458		

B. Transition to VxWorks

The transition from autonomy technology research to a realistic flight software environment involved at least three major phases. First was the re-development of applications, where needed, in the C or C++ programming language. The second phase was transition to the cFS environment under Linux, either in a Virtual Machine (VM) on a readily available office computing environment, either Microsoft ® Windows or Apple Mac ®, or on native Linux on Intel desktop or laptop computers. The transition to C/C++ and to cFS on Linux were generally conducted concurrently, rather than building applications as stand-alone command line executable applications before running within the cFS environment.

Once the applications were integrated and tested successfully in a Linux environment, the transition to VxWorks on an embedded Single Board Computer (SBC) was completed. The cFS environment was designed so that a properly built application that runs correctly on Linux will transition easily to a cFS environment on another operating system. The Operating System Abstraction Layer (OSAL) is a key to a smooth transition from Linux to VxWorks or other operating systems. Our experience generally validated the cFS design approach. Some of the applications executed correctly on VxWorks the very first time they were loaded.

With cFS, the details of the hardware and Operating System are abstracted in a Platform Support Package (PSP). So that cFS applications do not need to reference hardware-specific information. Memory interfaces, exception handling, timers, and file system information are encapsulated in the PSP to provide standard interfaces to other elements of the cFS. Prior development groups had built a PSP for the Power PC 750 board from Maxwell Technologies that we used as our primary VxWorks target. The PSP required very little customization, such as modifying tables that defined file system data locations to match the file system organization expected of the cFS software. VxWorks similarly uses a Board Support Package (BSP) that provides the detailed memory maps, drivers, interrupt handlers and other low level interfaces to the hardware. The BSP was provided by the vendor and was customized and tested by other development groups. With a completed BSP and PSP, we were able to build and load several of our applications easily.

Part way into loading and testing the application set, we found that the original VxWorks kernel that we obtained from the JSC development group was built without support for C++ libraries, and since several of our applications use libraries developed in C++, we needed to rebuild the VxWorks kernel with the needed features. The kernel build was reasonably straightforward, performed by developers with little prior VxWorks experience. Once completed, the remaining applications loaded and executed as well.

C. Integration Issues

Some of the issues that we encountered in transitioning to a flight-like architecture were:

- Command and telemetry message identification.
- Data byte order when using dissimilar processing architectures.
- Managing build configurations.
- Managing application data and models.

Command and Telemetry Definition Issues. Command and telemetry interfaces in cFS use a Software Bus with each interface specified as a message containing either a single command or a group of telemetry data values. Messages can be sent synchronously or asynchronously. Applications generate messages that are available for ingestion by any other application. The Message ID (MID) is the unique key for data access. The Software Bus uses ‘pipes’ to describe the destinations to which messages are sent. A pipe can be read by only one application, but each

application can read multiple pipes. Any message can be sent to multiple pipes, dependent on the number of applications that subscribe to a message. The Software Bus creates a routing table that contains information on what messages are to be sent to which pipes. Messages can be sent to processes on the same processor or on different processors, and applications do not need to maintain awareness of the destinations that will receive its messages – the routing table manages all the details based on applications creating messages and subscribing to them.

To assure uniqueness across multiple applications required central coordination between all applications and the messages to be generated. cFS message IDs are defined in software header (.h) files. Using common header files that are used by all applications assures uniqueness of the message IDs, but because applications can be added and moved readily, the header files must be updated to accommodate the changes. In a mature flight software environment the set of applications and messages passed between them is stable and rigorously defined by interface control documentation, data bases or other methods. In our transition environment we needed significantly more flexibility to change messages, move applications from one processor to another, remove and add applications, and modify the content of messages as needed. Managing the flexibility proved to be somewhat of a challenge, especially when executing multiple instances of an application on multiple processors. We resolved the issues using a set of rules for generating message IDs that used pre-defined ranges of allowable IDs for each application, coupled with a processor ID that was used by macros to generate guaranteed unique Message IDs regardless of configuration.

In addition to the messages generated and consumed by the applications development by our project, the Core Flight Executive (cFE) itself generates and consumes messages and commands. There are also a group of cFS "product line" applications that need unique message IDs. There can be multiple copies of these applications running on different processors or a single processor. Since these were built by other development organizations, their message IDs were already assigned, but were not assured of being compatible with the MID assignment decisions made by other users. There are three different sources of command and data messages:

- Core Flight Executive (cFE) messages.
- Core Flight System (cFS) Product Line Application messages. The cFS product line applications are developed for reuse by the cFS community. Commonly used functionality such as data logging, limit checking, application scheduling, and command generation.
- Project Application messages.

While a simple approach was appealing to a technology research and development group, it became difficult to manage without a coordinated and automated approach. The Command and Data Dictionary (CDD), mentioned above, was used to manage the application to application messages for the new applications being developed under the project. The application to application messages, however, were only part of the complete set of MID management. A system of software macros was eventually used that combined a processor ID, instance ID and message type that assured every message a unique value without conflict with any other message. While the approach successfully eliminated message conflicts, the value of the message ID was not readily apparent. Ongoing work will be needed to guarantee uniqueness of message IDs while making it easy for developers and systems engineers to readily interpret the message IDs used by each application in order to develop, integrate and test the integrated application.

Data Byte Order Issues. While cFS is designed to make the hardware architecture as transparent as possible to the applications that execute on them, the native architecture byte order is an attribute that is not fully handled by the cFS environment.

The interfaces between dissimilar processor architectures require proper handling of data byte order. cFS standardized messages using a CCSDS data format, with a primary header, secondary header and the message data, often referred to as the "payload". The primary header consists of the Message ID, a sequence number and the message size, and was by design always specified in big endian format. The secondary header includes a timestamp, and can accommodate additional customization by a project team. The secondary header byte order is left to the discretion of the project team and is generally specified in the processor architecture byte order. The message data is also at the discretion of the project, and also typically uses the architecture byte order. When cFS is used on a single processor, such as for a typical uncrewed satellite or planetary exploration spacecraft, the byte order decisions work well. In our distributed architecture with a mix of big endian and little endian processors, and with a goal of flexible, processor-neutral architecture, the byte order required close coordination, and changing the byte order at some points was required when exchanging data between processors.

The message data byte order was always left to the discretion of the application developers. Some applications generated messages in the generating processor byte order, requiring that a receiving application be aware of the byte order of the generating application. Including a byte order indicator as part of the data was one method that could be used to properly interpret the data. Some applications selected a byte order that would always be used,

defining a "wire byte order" of either big or little endian. When executing on a processor different from the specified byte order, the application would byte swap the data prior to use. The applications were assumed to have correct information about the data types in the message, since they intended to use the data for some purpose and would need to know how to interpret.

The final byte order element that needed byte order management is the secondary header. Our group chose to handle the secondary header in the SBN application, and selected a wire byte order for all inter-processor secondary headers. No action was needed for secondary headers between applications on the same processor, since the convention was to use processor byte order for the secondary header. Having SBN convert the headers to the wire byte order, and the SBN node on the receiving end convert from wire byte order to processor architecture byte order, we were able to achieve the goal of flexible and architecture-neutral applications.

The cFS provides a Data Store (DS) application that logs the messages to a data file for analysis or for data playback. The secondary header containing the timestamps depended on the byte order of the processor that recorded the data. To properly interpret the data required knowing what processor recorded the data, and if an application were moved from a big endian to a little endian machine, the data logs would have to be interpreted differently. A data delogger was used to convert the binary data logs to readable data formats, and the user needed to specify the byte order of the data logs. We added a byte order descriptor to the log file header to alleviate the problem of needing to know the timestamp byte order, particularly useful when reading a data file that may have been recorded months ago.

While byte ordering issues have been encountered, and solved, many times,[3] it continues to be an issue for cFS as well as for other mixed processor architectures. Either well-defined and specified protocols for handling byte order are needed, or a specification within the message is needed. Since bandwidth is precious, especially in space to Earth communications, the better approach is probably to specify a byte ordering policy rather than adding to the message size when the value will be unchanged once built and launched. For integration and testing it may be more convenient to transmit the byte order with the message, but the convenience could result in deferring decisions about byte ordering policy until the decision becomes more expensive.

SBN was designed to be unaware of message content, which requires that applications handle the byte order of their data. We have built architecture neutral applications so that an application can be seamlessly moved from one architecture to another without modification. The goal has been to simply recompile for a different target architecture and obtain the same behavior. We have freely moved applications from an i686 Linux, to PPC with VxWorks, in various combinations with excellent success. Architecture neutrality required that each application specify a "network byte order" for its data, so that regardless of the machine on which it is running, the transmitted messages are always in the same byte order. The transmitting application is responsible for converting to the network byte order, and receiving applications are responsible for converting from network byte order.

Managing Build Configurations. Using cFS and working with several applications integrated by different development groups posed several organizational and configuration management problems. The cFS source code is provided as open source software that enables customization by various development groups. Core cFS functions as well as a standardized reusable application set are deployed and controlled by a cFS community of users, and our development group is represented on the various cFS Control Boards that decide on the official software versions. Software is stored in repositories using the Software Configuration Management (SCM) tool known as *git*. *git* allows for individual developers or development groups to establish branches starting from known baselines to make changes for the specific needs, and our project made extensive use of branches to customize cFS applications to our needs. The applications that our group developed comprised a new *git* repository. We initially obtained the SCM controlled cFS *git* repositories and created project branches in which we could make modifications to customize to our needs. cFS uses a Platform Specific Package (PSP) in which the hardware and operating system customizations are contained, and while PSPs existed for Linux and VxWorks, we occasionally found that we needed to customize to the specific versions or configurations that we used. A Core Flight Executive (cFE) package contains the basic cFS functionality, including the Software Bus, Time Services, Executive Services, Event Services and Table Services. The core cFE software generally did not require customization of functionality, but some basic configuration settings are contained in the cFE Core software, such as Processor IDs, Processor Names, and settings for items such as clock controls, time stamp size and other global configurations. Some of these customizations applied to all cFS use across the project, but others, such as Processor ID settings, were based on a particular configuration needed by an individual developer for test and debug, or for various project configurations.

Unlike a typical flight program in which a specific architecture is selected early in design and all further development works toward that architecture, our work required flexible configurations. In development and test of an application, a developer can select the subset of interfacing applications, and execute them on a single processor to verify or debug an application's functionality. Later on the develop will add additional applications, and distribute

them among different processors. For example, a developer might test the ACAWS Fault Detector application with AMPS playback data running on a single Linux Virtual Machine. When successful, the developer could run the ACAWS Fault Detector on one Linux VM and play back data from a different Linux VM. Then the ACAWS Fault Detector could be moved to a Power PC running VxWorks while the AMPS playback is run from a Linux machine. Once ACAWS FD was successfully executed, the ACAWS Diagnostic Engine is added to the test configuration, and other applications as well. Each of these configurations is built from the same source repositories but requires build customizations.

Our solution was to use a set of build scripts that customizes the cFE core headers with the processor IDs and configurations, selected the applications needed for the build, managed the message and command IDs to assure that every application used unique IDs, and set definitions for byte order. Most of the configurations were customized with C source code header files. A configuration file, in a JSON format, specified the set of applications, processor IDs, processor type and other configuration information. The build scripts customized the message and processor IDs based on the build configuration to assure unique IDs.

When using multiple processors, the Software Bus is logically extended between processors with the SBN application, configured with a data table that defined the processor IP address and data ports to be used. Each execution using SBN required that each processor have the same SBN configuration, which required that the configuration be modified for the set of processors used for the run. Configurations ranged from single node runs, that did not require SBN, up to four processors, and several processors were available from which to select a configuration. We also had multiple networks that could be used. When using playback data, we could use a normal NASA site network, but when using the AMPS hardware rig, an isolated lab network was required. Because of the rather wide range of possible configurations, it was necessary to manually customize the SBN configuration to the specific configuration in use. The process was somewhat sub-optimal and potentially error-prone, but was workable for our integration setting. Scaling up to more processors will require improved management, but with maturity a program could be expected to lock down a set of test configurations that could simplify management of test configurations.

Managing Data and Models. Several data and model elements required configuration control and customization. The ACAWS application set is model-based using multiple data sets for each of the applications. Several data files were needed that were read at initialization. Using the same approach for a flight software environment as has been used on engineering workstations results in some difficulties. A flight system often does not contain a file system or disk drive, or provides a limited capability file system. When executing on a PPC with VxWorks, we used a RAM disk, which is an allocated block of memory with a file system implemented in memory. At bootup, the data files must be copied into the RAM disk which is completely erased when the processor is powered down. While we could generally manage to work with the limited file system, it was clear that scaling up to large-scale systems will require different approaches. For example, when completing the EFT-1 work in the coming months, there is a text file that is parsed at initialization time and loaded into data structures. An improved method may be to convert the data to its binary form off-line, and load the data directly into memory at initialization time. This is not presently a problem with the small size of the AMPS system model, but will be needed to scale to larger spacecraft system applications in the future.

Some of the applications use a set of data tables, using a Table Services feature provided by cFS. The cFS Limit Checker, Data Store and Application Scheduler all use Table Services for initialization and configuration data. As long as there is a single instance of the application, the approach works well. However, there were cases in which we needed to configure an application with different tables for different instances, such as the Application Scheduler. In these cases, the software CM system that we developed could be problematic because it was set up to host a single table in the repository. Modifications to allow per-instance data tables would improve the configurability and ease of use for these cases.

VI. Current Status

Several notable milestones were achieved by the project. The first major milestone was a demonstration of integrated applications running in the cFS environment under Linux. The degree of integration varied from data exchange between applications, to simply running in the same environment, on the same networks, without data collisions, corruption or system overflow. The ACAWS application with AMPS data was among the most tightly integrated group of applications. The AMPS system generated power system sensor data that was put onto the cFS Software Bus and transported via SBN and Ethernet to a second platform running the ACAWS Fault Detector application. The test results were put into another cFS message to transmit to the Diagnostic Engine, running either on the same platform or another machine. Several configurations were executed to confirm the flexibility and

architecture neutrality that was one of the objectives. The FD application was executed on the same platform as the Diagnostic Engine, exchanging messages on the Software Bus on a single processor, and was also executed on a separate Linux platform from the Diagnostic Engine, exchanging messages between processors of the same byte order, using SBN.

The second major milestone was the porting of applications from Linux to VxWorks, on an embedded platform. All applications have been executed on a Power PC 750 (PPC 750) processor with VxWorks. Both single-platform tests, with a set of applications on the PPC, and inter-processor configurations, with some applications on PPC and others on i686 with Linux, have been executed. The inter-processor tests confirm that the applications could communicate across different byte order. The ACAWS Diagnostic Engine application uses a commercial software library, TEAMS-RT, that required porting to VxWorks and PPC by the vendor, QSI. The initial port was a moderate complexity task that proceeded without major difficulty. The most complex aspect was in setting up a suitable environment that could support QSI as well as our application team and assure that QSI's development and testing results would transition to our project's use upon completion. We are continuing to integrate and test the integrated Diagnostic Engine in the PPC environment. Initial testing revealed some differences in outputs between correctly functioning applications on Linux and i686, and the VxWorks and PPC platform.

The HyDE and IMS applications executed successfully in the cFS environment on both Linux and VxWorks. Both applications ran with sample data provided by a recorded data playback application that was transmitted over the Software Bus to the diagnostic application. Results on both Linux and VxWorks were compatible indicating a successful transition.

The ACAWS applications were exercised with the AMPS system to confirm the correctness of the diagnostics and failure impact determination and interfaces to another automation application, the AMPS Power Controller (APC) developed at NASA Glenn Research Center. Faults are injected into the AMPS testbed at JSC, and the ACAWS application receives the sensor data. The Fault Detector application produces a set of Pass, Fail or Unknown test results and sends them to the Diagnostic Engine. The Diagnostic Engine makes a diagnosis of the failure modes or faulted components indicated by the test results vector from FD, and sends the results to the APC. The APC then uses the diagnoses to make decisions about reconfiguration to assure power to critical components, sending commands back to the AMPS testbed to turn off lower criticality equipment. An environment simulation controls a solar array simulator that requires the APC to account for the environment and configuration of the available power when determining the appropriate action when system faults occur. We demonstrate a scenario in which the spacecraft is in eclipse (no sunlight on the solar arrays) and a failure of the switch controlling battery inputs fails open. The only power remaining is one battery, so to assure power to the most critical loads until the spacecraft is in sunlight, the APC turns off lower criticality loads. The scenario demonstrates a fully integrated closed loop system in which the subsystems and their processors, a control system and a system-level diagnostic system collaborate to make critical autonomous decisions that would normally require crew and flight control teams.

While the effort to convert applications to cFS from applications in Java, Python and other languages was considerable, the transition from technology research to executing in a flight-like environment proceeded smoothly. Most of the development team had little or no experience with the cFS environment or embedded real-time software systems, and several had not programmed in C or C++ recently, so considerable work went into learning the environment while porting the software. Fortunately the team had several experienced cFS developers available for periodic consultation that eased the effort. The cFS API is not overly complex, and working with several good examples or templates made the effort of converting to cFS fairly straightforward.

Integrating applications developed by different groups, using different Software Configuration Management repositories, proved to be somewhat of a challenge. Each application group needed to use the same version of the cFS software and the applications that needed to interface together, particularly applications such as the SBN that was needed for inter-processor communications. Development groups at different NASA centers, while all using the *git* software CM system, had different legacy repository and build systems, and frequently it was difficult to assure or even determine if the software versions were compatible. Even though there was extensive planning, discussion and collaboration between groups about CM repositories, the desire to continue use of legacy systems continues to cause occasional issues. In retrospect, it probably would have been more efficient to decide on a common CM and software build system across all participants.

VII. Conclusions, Benefits and Future Use

The conversion of autonomy applications to cFS, VxWorks and PPC has clearly shown the feasibility of executing intelligent autonomy software in flight environments. We have shown that the software transitions to routinely used programming languages, C or C++, with relative ease and without any major dependence on features

not available in standard C language constructs. The ability to develop in Linux, and conduct initial testing, and then move to the target environment, was largely validated; as one of the primary benefits of cFS, we expected to validate the ease of transition between operating systems, and were not disappointed. But perhaps the most important accomplishment that we have achieved is the creation of a test and evaluation environment in which to examine in depth some important questions about spacecraft autonomy applications.

With the set of applications, processor and software environments, and the system testbeds available to us, we are fully configured to conduct extensive performance testing and analysis. The performance analysis of autonomy applications is needed before flight programs can be expected to build autonomy into future spacecraft. First, they need clear autonomy requirements for the classes of missions to be flown. Second, flight programs must be able to accurately estimate the processor performance required of a full-scale autonomy system in order to correctly establish the number of processors, their memory and data storage needs, network bandwidth and the resultant mass, power and thermal impacts of the processors required for the autonomy applications. And third, information about the development effort required for new classes of autonomy applications is needed to mitigate the cost and schedule risk of including autonomy applications that have not been previously built and flown at scale.

The initial focus will be on determining the processor, memory and communications bandwidth of the applications. The data models exercised to date are relatively small compared to the expected needs of a Mars or Lunar environment spacecraft, but serve as a good starting point that will help establish a minimal baseline. There is always some "idling" resource consumption that must be characterized, which will be among the first performance measure taken.

The ACAWS application has previously been executed with the Orion spacecraft, most notably conducting a flight following exercise during the EFT-1 flight test in December 2014[15]. The model is substantially scaled up from the AMPS model; AMPS contains about 70 failure modes, while the EFT-1 model contains nearly 3500 failure modes and 2400 tests. We are working to execute the EFT-1 model in the cFS system on a PPC 750 that will provide the ability to measure performance in a scaled up execution environment. We will be able to compare the idle performance, the AMPS model and the EFT-1 model to gather substantial information about the resource needs of each model, which will enable an extrapolation of the scaling factors. Using the number of failure modes as a size determinant, we want to be able to accurately estimate the performance requirements of larger models so that a future program can estimate the number of failure modes that a spacecraft will have and know how much resources must be allocated in order to support the diagnostic system. The ACAWS application will be the first application for which scaling metrics will be determined. Other applications for state estimation, planning, procedure automation and other autonomy applications will follow, based on procedure counts, the number of decisions to be automated or other suitable measures. The objective will be to determine a set of parametric values from which the autonomous application sizing estimates can be made, and the program will have a good estimate of the processor performance requirements for the system.

Future work will also expand the HyDE model and use the MSFC capabilities to connect to live data from the PPA test rig, monitor system performance and fine tune the PPA fault models. We will extend monitoring and fault diagnosis to the Sabatier reactor, which produces CH_4 as input to the PPA. Finally, we will integrate fault management across the power and air-side ECLSS system. This final task poses a variety of challenges; not only will the complete spacecraft system model be significantly larger than any individual model, but the interaction of faults across disparate subsystems will push both the processor load and also the knowledge and modeling effort needed to ensure that all fault conditions are correctly identified.

Once the performance parametrics are determined, our work will help to guide programs in the determination of autonomy requirements. Autonomy requirements will need to specify an autonomy duration, that is, how long the spacecraft and crew must operate entirely without ground assistance. When light-time constraints come into play for long-distance solar system missions, an absolute minimum will be the round trip light time. Of course, a reaction time will also be needed, so for a Mars mission, with one way light times over 20 minutes when Mars is at opposition, the spacecraft can't possibly get support from Earth in less than 40 minutes. Conceivably, a program could identify all faults that require decisions in less than 40 minutes and build autonomy for just those failure cases, and in all other cases put the spacecraft into some type of "safe mode" and use a large ground support workforce to determine actions for other failures. However, if autonomy requirements are that the spacecraft must be able to return from any point in the mission entirely without support from Earth, the autonomy system will need to look substantially different. The autonomy performance measures that will be taken with the systems we have developed should be able to provide valuable information about both the need for and the achievability of a range of possible autonomy requirements.

In addition to system performance and requirements, programs will need information about the development effort and cost of building unfamiliar new technology and the data models that execute on the applications. This is a

more difficult estimation activity, but one in which at least some preliminary information is available. Since our project has not had to follow rigorous design, development and test processes, we are not able to provide accurate data about the complete effort involved in building a model-based decision-making system. We do have at least general information about how much effort has been expended on the models that we have built, and with the framework that we have put in place, future maturation projects can use the system to better refine development cost estimation methods.

While the path from developing intelligent automation technology to deployment in spacecraft is a long one, our work has made significant strides toward adapting technology concepts to execute in full scale flight avionics environments and the creation of an environment in which to continue to develop, analyze and evaluate both the technology needs and performance environments for full scale autonomy needed as human operations into the solar system are planned and developed.

Acknowledgements

The authors thank the project team that achieved so many successes working in new environments with unfamiliar tools. Specifically, Chris Knight, Mike Scott and Keith Swanson led the way in transitioning to cFS and VxWorks; John Ossenfort, Adam Sweet and Vijay Baskaran built ACAWS in the cFS environment. Numerous people from other NASA centers provided critical help at important times. Pat Castle and Scott Christa at ARC, and Steve Duran at JSC, were critical to getting started with VxWorks. Jim Ratliffe, Danny Carrejo and their team at JSC made sure that we had the integration environment in the iPAS that we needed. Pat George, Anne McNelis, Billy Hau at GRCC helped us understand and integrate with the AMPS system and gave us challenging diagnostic scenarios. Morgan Abney and Zach Greenwood at MSFC provided assistance with PPA datasets and fault identification expertise. Richard McGinnis at NASA HQ was our advocate and guide in developing the project's goals and kept us on track. This project was funded by the NASA Advanced Exploration Systems Program.

References

[1] J. Frank, L. Spirkovska, R. McCann, L. Wang, K. Pohlkamp, L. Morin. Autonomous Mission Operations. Proceedings of the IEEE Aerospace Conference, March 2-9, 2013.

[2] H. Stetson, J. Frank, A. Haddock, R. Cornelius; L. Wang; L. Garner. AMO EXPRESS: A Command and Control Experiment for Crew Autonomy. Proceedings of the AIAA Conference on Space Operations, September 2015.

[3] R. Cornelius and J. Frank. International Space Station (ISS) Payload Autonomous Operations Past, Present and Future. Proceedings of the AIAA Conference on Space Operations, Aug. 31 – Sept. 2, 2016.

[4] J. Frank, D. Iverson, C. Knight, S. Narasimhan, K. Swanson, M. Scott, M. Windrem, K. Pohlkamp, J. Mauldin, K. McGuire, H. Moses. Demonstrating Autonomous Mission Operations Onboard the International Space Station. Proceedings of the AIAA Conference on Space Operations, Aug. 31 – Sept. 2, 2015.

[5] D. McComas, J. Wilmot, A. Cudmore. The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft. 30th Annual AIAA/USU Conference on Small Satellites, Logan, Utah, August 6-11, 2016.

[6] Z. Greenwood, M. Abney, J. Perry, L. Miller, R. Dahl, N. Hadley, S. Wambolt, and R. Wheeler. Increased Oxygen Recovery from Sabatier Systems Using Plasma Pyrolysis Technology and Metal Hydride Separation. 45th International Conference on Environmental Systems, Bellevue, Washington, July 12-16, 2015.

[7] R. McCann, L, Spirkovska, and I. Smith. Putting ISHM Capabilities to Work: Development of an Advanced Caution and Warning System for Crewed Spacecraft. AIAA Modeling and Simulation Technologies (MCT) Conference, August 19-22, 2013.

[8] A. Mathur, S. Deb, and K. Pattipati, "Modeling and Real-Time Diagnostics in TEAMS-RT," Proc. American Control Conf., IEEE Press, 1998, pp. 1610–1614.

[9] P. Morris, M. Do, R. McCann, L. Spirkovska, M. Schwabacher, J. Frank. Determining Mission Effects of Equipment Failures. Proceedings of AIAA Space, Aug. 4-7, 2014.

[10] S. Narasimham and L. Brownstone. HyDE - A General Framework for Stochastic and Hybrid Model - Based Diagnosis. Proceedings of the 18th International Workshop on the Principles and Practices of Diagnosis, 2007, pp. 162 - 169.

[11] D. Iverson. Inductive System Health Monitoring. Proceedings of the International Conference on Artificial Intelligence, 2004.

[12] R. May, J. F. Soeder, R. F. Beach, P. J. George, J. Frank, M. A. Schwabacher, L. Wang and D. Lawler. An Architecture to Enable Autonomous Control of Spacecraft. AIAA Propulsion and Energy Conference, July 28 – 30, 2014.

[13] J. Soeder, T. Dever, A. McNelis, R. Beach, L. Trase, R. May. Overview of Intelligent Power Controller Development for Human Deep Space Exploration. 12th International Energy Conversion Engineering Conference (IECEC), Cleveland, Ohio, July 28–30, 2014.

[14] OSAL Library API.doc, https://github.com/nasa/osal/blob/master/doc/OSAL%20Library%20API.doc

[15] G. Aaseng, E. Barszcz, H. Valdez, H. Moses. Scaling Up Model-Based Diagnostic and Fault Effects Reasoning for Spacecraft. Proceedings of the AIAA Conference on Space Operations, Aug. 31 – Sept. 2, 2015.