Propulsion IVHM Technology Experiment Overview

Claudia M. Meyer
NASA Glenn Research Center at Lewis Field
21000 Brookpark Road
Cleveland, OH 44135
(216) 977-7511
Claudia.meyer@grc.nasa.gov

Howard Cannon
NASA Ames Research Center
Moffett Field, CA 94035

Edward Balaban
QSS Group
NASA Ames Research Center
Moffett Field, CA 94035

Chris Fulton, Bill Maul and Amy Chicatelli
Analex Corporation
NASA Glenn Research Center at Lewis Field
Cleveland, OH 44315

Anupa Bajwa
USRA-RIACS
NASA Ames Research Center
Moffett Field, CA 94035

Edmond Wong
NASA Glenn Research Center at Lewis Field
Cleveland, OH 44135

Abstract—NASA researchers recently demonstrated successful real-time fault detection and isolation of a virtual reusable launch vehicle main propulsion system. Using a detailed simulation of a vehicle propulsion system to produce synthesized sensor readings, the NASA team demonstrated that advanced diagnostic algorithms, running on flight-like computers, can, in real time, successfully diagnose the presence and cause of faults. This demonstration was conducted as part of the NASA Propulsion IVHM Technology Experiment, or PITEX.

INTRODUCTION

The Propulsion IVHM Technology Experiment (PITEX) is a NASA effort being conducted cooperatively by NASA's Glenn Research Center, Ames Research Center and Kennedy Space Center. It is a key element of a Space Launch Initiative (SLI) Risk Reduction Task being performed by the Northrop Grumman Corporation in El Segundo, California. PITEX has several main objectives. First is the continued maturation of diagnostic technologies that are relevant to 2nd Generation Reusable Launch Vehicle (RLV) subsystems. Second is an assessment of the real-time performance of the PITEX diagnostic solution. Third is the migration and evaluation of the PITEX diagnostic solution in Northrop Grumman's Integrated Vehicle Health Management (IVHM) Virtual Test Bed (IVTB). In the IVTB, a broad range of vehicle subsystem health managers, in addition to propulsion, will be considered, and the benefits of coordinating the subsystem health managers through area and system-level health managers will be demonstrated. PITEX is laying the groundwork for future subsystems.

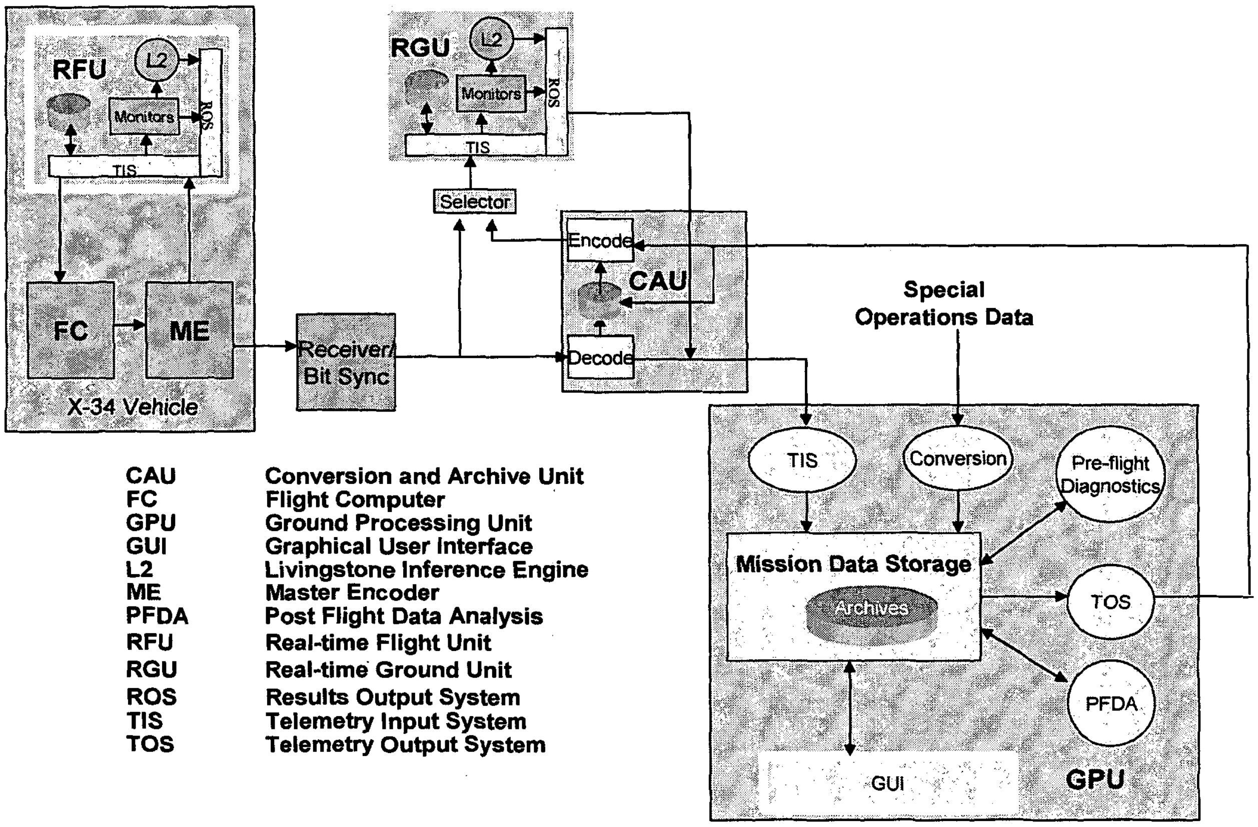
The current PITEX effort has considerable legacy in the NASA IVHM Technology Experiment for X-vehicles (NITEX), selected to fly on the X-34 sub-scale RLV that was being developed by Orbital Sciences Corporation. NITEX, funded through the Future-X Program Office, was to advance the technology readiness level of selected IVHM technologies within a flight environment, and to begin the transition of these technologies from experimental status into RLV baseline designs. The experiment was to perform real-time fault detection and isolation and suggest potential recovery actions for the X-34 Main Propulsion System (MPS) during all mission phases using a combination of system-level analysis and detailed diagnostic algorithms. In addition, the experiment was to demonstrate the use of an advanced, user-friendly ground station that combines information provided by the on-board IVHM software with information obtained while the vehicle was on the ground.

This paper describes the original architecture that was designed to meet the NITEX objectives. The particular portions of this architecture that were implemented and subsequently demonstrated under PITEX are discussed in detail. The X-34 MPS and associated simulated failure scenarios are described. Finally, the metrics that were collected during the testing of the PITEX diagnostic system on flight-like hardware are discussed and results are presented.

NITEX ARCHITECTURE

The NITEX architecture shown in Figure 1 was designed to support fault detection and isolation of the X-34 MPS throughout all mission phases. The experiment was to act as an advisory system for detecting both functional failures that can impact the current mission as well as degraded component performance that may impact future missions. In order to accommodate the various mission phases (primarily propellant loading and ground checkout, captive carry on the L1011 and powered flight), the architecture had both on-board and ground-based components. The experiment was to receive MPS sensor and command data

[1] U.S. Government work not protected by U.S. copyright.
[2] Paper number 1481



CAU	Conversion and Archive Unit
FC	Flight Computer
GPU	Ground Processing Unit
GUI	Graphical User Interface
L2	Livingstone Inference Engine
ME	Master Encoder
PFDA	Post Flight Data Analysis
RFU	Real-time Flight Unit
RGU	Real-time Ground Unit
ROS	Results Output System
TIS	Telemetry Input System
TOS	Telemetry Output System

Figure 1 - NITEX Architecture

in order to track the state of the components, detect off-nominal conditions, isolate failures to individual components, and, when appropriate, recommend a recovery response.

These tasks were to be performed on a NASA-provided avionics box located on the vehicle (Real-time Flight Unit or RFU) or a ground-based commercial grade version of the avionics box (Real-time Ground Unit or RGU) and a ground-based health monitoring station (Ground Processing Unit or GPU). The RGU and RFU hosted identical diagnostic software. The RGU was intended to support, in real-time, those missions on which the RFU was not manifested on the X-34 vehicle and to support mission playback. The GPU was responsible for performing all monitoring while the vehicle was on the ground. During flight, high-level status information was to be telemetered to the ground processing unit from the avionics box. The ground station was to provide both high-level status information as well as more detailed analyses. These more detailed analyses included mission-to-mission trending.

Since the cancellation of the X-34 program, the software items in Figure 1 that are enclosed in bold yellow (on the RFU, RGU and GPU) have been implemented and demonstrated on relevant hardware. These software components are addressed in more detail in the next section.

PITEX DEMONSTRATION ARCHITECTURE

The overall PITEX demonstration system architecture is shown in Figure 2. The demonstration system is designed to test a subset of software components from the NITEX flight experiment architecture. The demonstration system features simulated propulsion system data being processed by diagnostic software on an RGU and display software hosted on a GPU.

Hardware

The PITEX diagnostic software resides on a Radstone PPC4A-750 VME single Board Computer. The card is housed in a chassis with a VME backplane and SCSI hard drive. I/O ports provide both serial and ethernet accessibility to the card. The PITEX diagnostic software is compiled in the Tornado II VxWorks (release 5.4) environment using the compiler supplied with Tornado. The VxWorks kernel is based on the Radstone board support package release 1.2/1.

The GPU hardware is a personal computer with a Pentium III 550 MHz processor, 256 MB of RAM, and 30 GB of hard disk storage. The GPU uses the Linux Operating System Redhat version 6.2 and communicates to the RGU through a TCP/IP Ethernet connection.

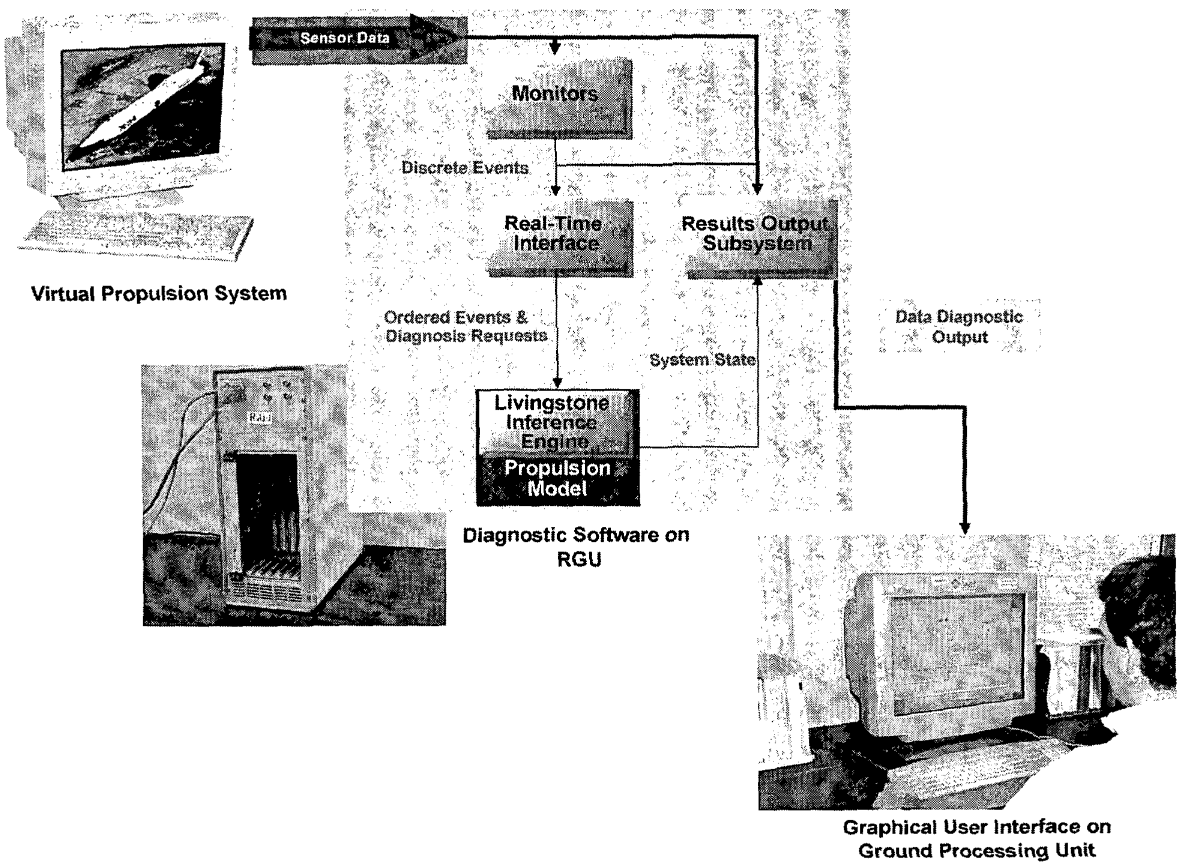

Sensor Data
Monitors
Discrete Events
Real-Time Interface
Results Output Subsystem
Virtual Propulsion System
Ordered Events & Diagnosis Requests
System State
Data Diagnostic Output
Livingstone Inference Engine
Propulsion Model
Diagnostic Software on RGU
Graphical User Interface on Ground Processing Unit

Figure 2 - PITEX Demonstration Architecture

Software

The key software components of the PITEX demonstration system include the Virtual Propulsion System, the Telemetry Input System (TIS), the Monitors, the Real-Time Interface (RTI), Livingstone, the Results Output System (ROS) and the Graphical User Interface (GUI). The TIS, Monitors, RTI, Livingstone, ROS and Virtual Propulsion System data sets reside on the RGU. The GUI resides on the GPU.

Virtual Propulsion System—This support component provides simulated data of the physical system for evaluation of the X-34 MPS feed system and for verification and validation of the diagnostic software. No system-level testing of the X-34 MPS was performed, making these simulated data the only data available.

The specific details of every X-34 subsystem and component will not be addressed in this paper; a history of the vehicle and an overview of its main propulsion system can be found in the literature [1, 2]. The selected Design Reference Mission for the X-34 MPS was the captive carry mission phase. The captive carry mission phase was selected due to crew safety considerations of the piloted L-1011. During this phase of operation, the X-34 is carried to the required launch altitude of 38,000 feet while it is attached to the underside of an L-1011 aircraft. The engine is not firing, and most of the other subsystems of the MPS are in a quasi-static state, except for the liquid oxidizer (LOX) and RP-1 (fuel) subsystems.

Throughout the first half-hour of captive carry, the MPS is locked-up. After this lock-up phase, the vent/relief system is activated to provide LOX conditioning. For two hours, this process maintains, within pre-defined thresholds, the nominal temperature and pressure in the LOX tanks. Once this is completed, the pressurization system is enabled. Subsequently, RP-1 bleed is performed for three minutes. During this process, fuel flow to the engine is maintained at a small rate, which is much less than that required for nominal operation of the engine. Next, LOX chilldown and bleed are performed to cool the warm feed line from the MPS to the engine, and to prepare the engine for powered flight. Nominally, LOX chilldown and bleed span six minutes, and occur at the end of captive carry.

The Virtual Propulsion System component includes both a Rocket Engine Transient Simulator (ROCETS) model and MATLAB routines that predict the behavior of various

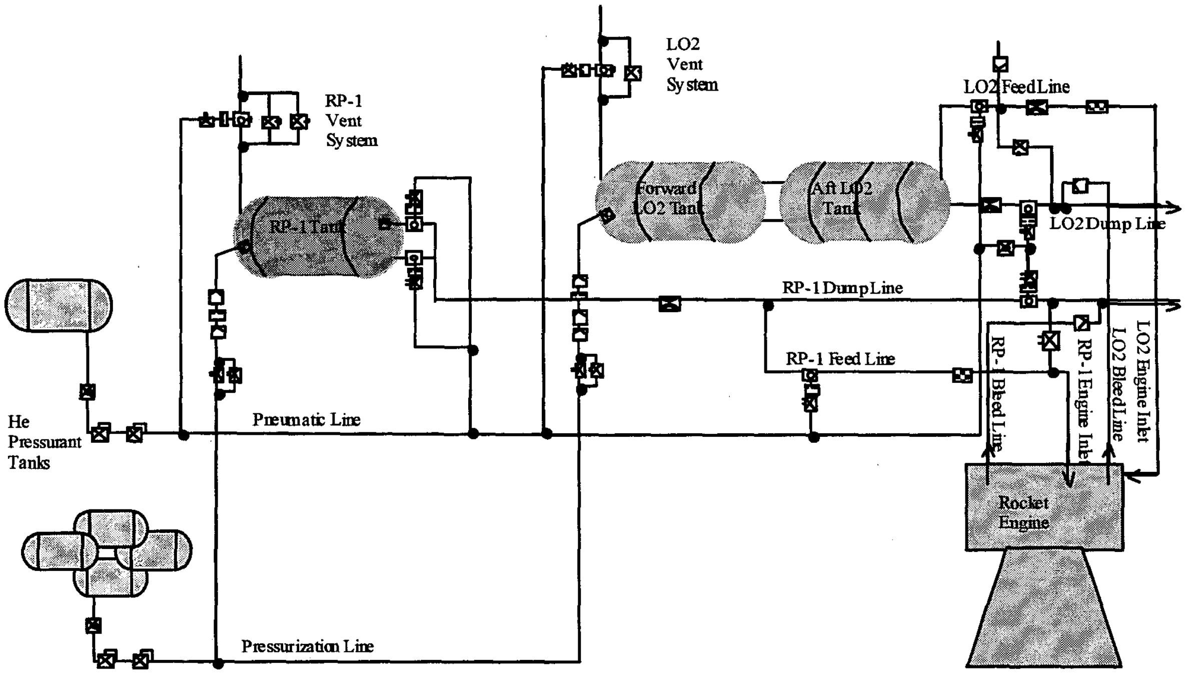


Figure 3 - X-34 Main Propulsion System Schematic

portions of the physical system during different modes of operation (i.e., propellant conditioning and bleed). The ROCETS program was selected because of its proven ability to reliably simulate large rocket propulsion systems. The physical scope of the model extends from the helium bottles, through the propellant tanks, to the ullage venting system and LOX and RP-1 feedline and dump-systems; all of these elements are shown in the X-34 MPS schematic displayed in Figure 3. The ROCETS model does not include the purge, reaction control, or pneumatic systems. The ROCETS MPS model was initially developed to simulate delivery of propellants to a LOX/RP engine during powered flight. It is not well-suited for simulating LOX conditioning since the dynamic behavior of the LOX tanks during this period is substantially different than during steady-state powered flight. Therefore, a MATLAB code was used to simulate LOX conditioning. The ROCETS/MATLAB models produce output files with time histories of selected parameters.

The virtual MPS provides the capability to study both nominal and off-nominal behavior of the X-34 MPS feed system. Several off-nominal scenarios have been simulated. These include valves sticking closed or open, valves spontaneously closing or opening, regulator failures, and sensor and microswitch failures. The virtual MPS can also generate parameter traces indicative of more subtle degradations such as the clogging of a filter, a small obstruction in an orifice, or degradation in valve actuation.

The accuracy of the data sets generated by the virtual MPS is dependent upon the approximations made when defining the components and the physical processes. Because of this, there is some amount of modeling error present in the results. These types of errors can be substantially reduced by obtaining accurate component and system-level data and anchoring the simulations with these test data. Although this step in the diagnostic system development process was not possible in the case of the X-34, the data generated by these simulation models permit appropriate initial testing of the diagnostic software.

Included in the Virtual Propulsion System component is a utility which combines the various output files into one flight-like data set. This utility adjusts the output data from the simulations to correspond to sensor locations, applies random noise to the data and inserts the discrete signals (i.e. commands and switch indicators). It also adjusts the sampling rates output by the models to correspond to the expected telemetry rates. The entire data set generation process is summarized in Figure 4. The final data set is in a standardized binary format, containing header information that declares the file's content and creation date.

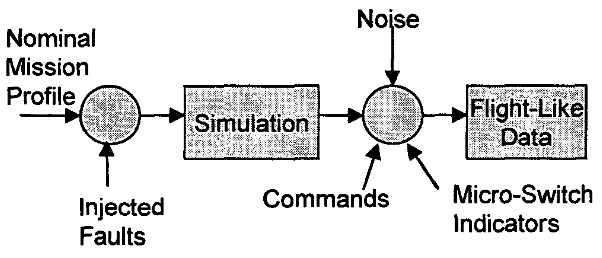


Figure 4 - Scenario data set generation process

Telemetry Input System—The demonstration TIS provides the interface between the flight-like data set and the Monitors and has three basic functions. First, the TIS provides an accurate 80 millisecond timer to simulate the hardware interrupt that would occur when a frame of telemetry becomes available. Second, when the simulated interrupt occurs, the TIS reads the next frame worth of data and stores it within an internal data buffer. The buffer currently holds ten seconds worth of data, but the capacity can be changed based upon specific memory limitations. Finally, data buffer access is provided to the diagnostic system by a TIS library routine. Access of a specific frame of data can commence after the TIS sends a notification to the Monitors that data are available. The TIS is designed to guarantee that the simulated telemetry data are available to the diagnostic system in real time.

Monitors—The Monitors are a collection of software functions that receive incoming propulsion system data, perform specific processing on these data, and then provide the resulting observations to the Real-Time Interface (RTI) component in the form of inter-process messages. Various types of system data are processed including the discrete command and switch indicator signals and the raw digitized performance sensor signals.

For every sensor signal, the Monitors consider a predefined number of bands that span the sensor's performance range. During operation, the Monitors examine each sensor data stream and continuously qualify the values with respect to these discrete bands. A sensor is determined to reside in a particular band when it satisfies a persistence criterion, that is, when a certain number of recent values have fallen within the bounds of that band. This qualitative assignment is then passed on to the RTI as a discrete event for use in subsequent system diagnosis.

In many failure scenarios, the rate-of-change for certain sensor data are key elements to be considered. As a result, the Monitors continuously calculate these rates and then qualitatively identify them in a manner similar to that described in the previous paragraph for sensor range checks. Figure 5 illustrates this sequence of operations. First the signal is processed to obtain the rate-of-change time history; this is followed by the qualitative identification. The rate-of-change calculation is performed by line-fitting a user-defined window of averaged data points over the corresponding time interval. This computation is repeated for successive windows of data resulting in a stream of rate values. During periods of rapid transition in the sensor data, the Monitors shift into a "fast rate mode" to more effectively capture these transients. In this mode, successive data points, rather than larger windows of points, are used to compute the slope.

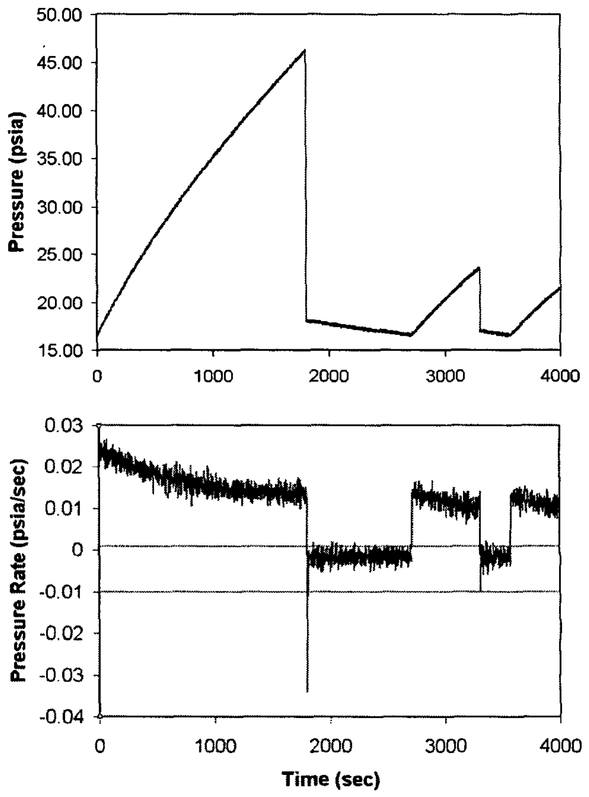


Figure 5 – Illustration of the rate-of-change monitor processing pressure data into qualitative rate information

To ensure robustness against signal noise, the Monitors implement a smoothing operation on the raw sensor data. Prior to the aforementioned sensor processing and rate calculations, the incoming data are first smoothed to attenuate noise that is present in the signals. The smoothing method involves the calculation of the average value of the incoming data stream over a predefined number of data points. For each successive calculation, the window of data points is moved incrementally forward in time by a pre-defined time-step. As each average value is calculated, it is time-stamped with the time associated with the end point of the averaging window.

Additionally, the Monitors include a Redundant Channel Comparison Module (RCCM) which provides qualitative information about the difference between two redundant sensors. This module was implemented to address failure scenarios involving redundant control loop sensors. The RCCM derives absolute difference values between two given sensor values; these resulting "deltas" are then qualitatively identified as well.

One final Monitor function is to provide timing information in response to timer request messages from the RTI. These requests are processed, and expiration notifications are returned when all Monitor processing is complete for the telemetry frame in which the timer expires. This ensures the timeliness of all reported observations required by Livingstone when making diagnoses.

Real-Time Interface—The purpose of the Real-Time Interface is to transmit discrete events from the Monitors to the Livingstone inference engine. To do this, the RTI handles four basic tasks. First, it translates the Monitor data into a format that is understood by the Livingstone model. Second, it uses timing information associated with events to package the information into discrete Livingstone time steps. Third, it decides when to request a diagnosis from Livingstone. Finally, it dictates when Livingstone information is transmitted to the GPU via the Results Output System (ROS).

To implement this functionality, the RTI was designed with two primary classes. The first is a translator class which is used to translate the Monitor data into Livingstone model variable information; this is largely a bookkeeping task. The second class is the RTI policy, which decides how to package the discrete events into time steps, when to request diagnoses, and when to transmit information to the ground. This policy has evolved throughout the NITEX/PITEX development and testing phases. The three key needs/assumptions driving the policy design are described along with the current design.

When a controller command occurs, there is a finite amount of time in which the system will stabilize in response to the command, and this amount of time is known a priori for each command type. Likewise, when a spontaneous failure of a component occurs, there is a finite amount of time in which the system will stabilize in response to the failure, and this amount of time is also known a priori for each component. These periods of time are known as latency periods, and they are the primary driver in determining diagnostic delay. To handle this, the RTI policy issues a timer request corresponding to the latency period after a command or spurious observation occurs. While the timer is pending, the RTI buffers the observations that are received; later observations override prior observations. When the timer expires, the RTI sends the event along with all of the latest observations to Livingstone, and then requests a diagnosis.

Controller commands can occur very rapidly, so that sufficient time does not exist for the effects of one command to completely settle before another command is issued. The RTI policy should unconstrain information that has had insufficient time to settle during these rapid periods, but it should try to maximize the evidence that is provided to Livingstone for diagnosis. The current policy for handling these "overlapping events" is to divide time into a number of segments as shown in Figure 6. When a command arrives, a time segment is created which stores the event and associated observations. If another command arrives that overlaps the transient period of the first command, two additional time segments are created. When Time Segment I is debuffered and sent to Livingstone, all of those observations that have not had sufficient time to settle and that belong to the same subsystem as command 1 are set to "unknown" (i.e., unconstrained). When Time Segment II is debuffered, only the observations in the same subsystem as command 2 are unconstrained (because the observations related to command 1 have now had time to settle). Finally, when Time Segment III is debuffered, none of the observations are unconstrained. Although a single overlapping event is used for illustration purposes, the policy handles any number of overlapping events, and considers cases where the transient periods are not equal.

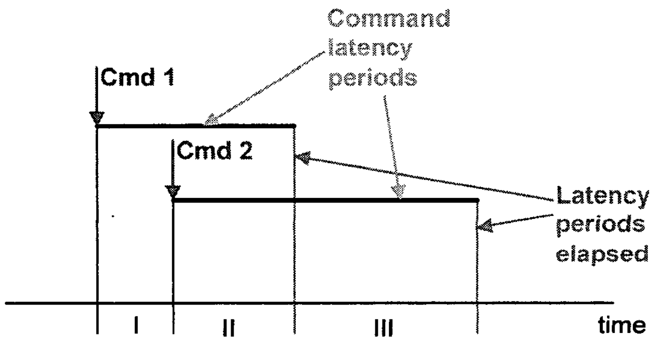


Figure 6 - RTI Policy for handling overlapping events

Monitors reside in multiple tasks. Therefore, their outputs are not necessarily temporally synchronized. The RTI policy must ensure that a time segment is not debuffered prior to all of the observations that belong to that segment being posted. The RTI is capable of ordering Monitor observations and requesting diagnoses appropriately with the guarantee from the Monitors that, following a timer expiration, no events will be sent to the RTI with time tags prior to the expiration time. Therefore, whenever the RTI policy receives a timer expiration, it is safe to debuffer all of those time segments with end times prior to the expiration time.

Finally, current system needs dictate that faults be detected as soon as possible. Therefore the current policy is to request and transmit the diagnosis after every time segment is debuffered.

*Livingstone—*Livingstone is the diagnostic engine responsible for inferring the health of the MPS. It does this by utilizing a high-level declarative model of the propulsion system and discretized sensor and command information (event data) generated by the Monitors. As event data are received from the Monitors (through the RTI), Livingstone continually updates its belief regarding the state of the various components in the system. The model is used to determine the expected observations given the component state. When there is a discrepancy between the expected observations and the actual observations, Livingstone searches for the most likely set of component states/failures that could produce the observations. Livingstone can also generate recommended recovery actions; however, for this demonstration it was used only for detection and isolation of failures. More information on Livingstone and the X-34 MPS Livingstone model can be found in [3,4].

*Results Output System—*The Monitors and Livingstone pass messages to the Results Output System (ROS) for transmission to the GPU. The source task controls the routing of messages, specifying local storage, 'downlinking', or both.

*Graphical User Interface—*The purpose of the GPU Graphical User Interface (GUI) is to display diagnostic information from the RGU or RFU to ground personnel who would be monitoring the system. The GUI has been designed for use by ground operators who are primarily concerned with the status of the vehicle, and experiment developers who are interested in what is going on behind the scenes in the experiment. Therefore, the GUI has been designed with the idea that it should be relatively intuitive and easy to use, provide quick access to raw data so the ground operator can judge the correctness of a diagnosis, and provide sufficient information to understand why the diagnostic system is making a particular diagnosis.

The GUI contains both a timeline view and a schematic view. The timeline view has indicators that are highlighted in red when a failure occurs to alert the operator. Strip charts in the timeline view display the continuous engineering unit data, Monitor readings, and the values of selected Livingstone model variables over time. Also in this view is a list of failure candidates with their identified rank (related to probability) and associated failed components. The schematic view highlights implicated components in color when a specific failure candidate is selected from the list in the timeline view. Furthermore, when the cursor is placed on top of a component, the component's current readings are displayed.

The GPU GUI software has been written in Java so that it is portable across multiple platforms. It can be used to replay previous missions or to connect directly to the RGU for monitoring an active mission. The GPU GUI software was designed in such a way that it may be easily reused when deploying Livingstone on other subsystems or applications.

HARDWARE TESTING

Client System Scenarios

The X-34 MPS simulation described previously was exercised under both nominal and off-nominal conditions during the captive carry mission phase. The resulting data sets were used to test the diagnostic software. Table 1 summarizes the failure scenarios that were considered in the PITEX demonstrations. All simulated faults occurred during the captive carry mission phase; both LOX and RP-1 subsystem faults and single and double faults were considered. Each bulleted item in Table 1 represents a distinct failure scenario. In some cases, several failure modes are considered for the same component. Some clarification with respect to failure scenario naming is in order. The difference between a valve sticking closed and failing closed, for example, is with respect to the valve's operation prior to the time that the failure occurred. The LOX Vent/Relief Pneumatic Pilot valve 'fails closed' during LOX conditioning at a time when it has been and should continue to be open. This same valve 'sticks closed' at a time during the LOX conditioning phase when it is commanded open following an interval in which it was commanded closed. Analogous descriptions apply to valves sticking open and failing open.

Table 1 – X-34 MPS failure scenarios used in PITEX demonstrations

Mission Phase	Component Failure
LOX conditioning	• LOX vent/relief pneumatic pilot valve sticks open • LOX vent/relief pneumatic pilot valve fails open • LOX tank vent/relief valve sticks open • LOX tank vent/relief valve fails open • LOX vent/relief pneumatic pilot valve sticks closed • LOX vent/relief pneumatic pilot valve fails closed • LOX tank vent/relief valve sticks closed • Open switch on LOX vent/relief pneumatic pilot valve fails • Open switch on LOX vent/relief pneumatic pilot valve fails and pneumatic pilot valves fails • Two Forward LOX tank ullage pressure sensors fail low • Two Forward LOX tank ullage pressure sensors fail high
LOX chilldown/bleed	• LOX feed pneumatic valve sticks closed

Mission Phase	Component Failure
	• LOX feed pneumatic pilot valve fails closed • Primary pressurization regulator fails low • LOX tank primary pressurization valve sticks open • LOX tank primary pressurization valve sticks closed • Closed switch on the LOX feed pneumatic valve fails
RP-1 bleed	• RP-1 feed pneumatic valve fails closed • RP-1 vent/relief valve fails open • RP-1 tank primary pressurization valve sticks open • RP-1 tank primary pressurization valve sticks closed
RP-1 bleed and LOX chilldown/bleed	• Primary pressurization regulator fails high • Primary and secondary pressurization regulators fail high

Metrics

There were four objectives in testing the PITEX diagnostic software on flight-like computer hardware. The first involved validating the correctness of the results. Second, timing information was extracted from the results with the goal of measuring PITEX diagnostic speed. Timing assessments were made with respect to various PITEX modules; however, the overall time required from the first time at which a fault can be sensed until the time at which the diagnostic system reports the correct diagnosis was of primary interest.

Third, resource utilization assessment was a key objective of the hardware testing. Memory and CPU usage were extracted and analyzed. The memory requirements for the PITEX software were established in order to infer memory requirements when expanding the software for a given application or to cover a new application. Static, stack and dynamic memory usage were assessed on a per task basis and overall system memory usage was determined. Average and maximum CPU usage were collected for each scenario, nominal and failure, over the entire scenario length. These statistics were collected on a per task basis and summarized for the overall software system as well.

Finally, measurements that are indicative of telemetry bandwidth of a flight experiment were collected. Although the current PITEX code does not have an efficient telemetry representation, maximum and average telemetry per second rates were projected based on a more compact integer representation. These numbers were analyzed with respect to Livingstone diagnoses.

RESULTS

When the scenario data sets summarized in Table 1 and the nominal scenario were run through the PITEX diagnostic software, the correct diagnosis was achieved in all cases. In some cases, several fault candidates were proposed. For example, when the LOX feed pneumatic valve sticks closed, PITEX attributes this to either a failure in this valve or the LOX Feed Pneumatic Pilot Valve – both faults have equal rank. This is because there is insufficient instrumentation to distinguish between these two failure modes. In all cases, however, the fault candidates included the actual failure mode.

For the failure scenarios considered, the overall diagnostic delay was found to be 20 seconds on average. The primary factor in the diagnostic delay is the latency timeout period applied. The timeout delay was instituted for a variety of reasons. First, since Livingstone works with qualitative propositional logic, it is not designed to handle transient states of the system. Therefore, the physical system settling time – the time required for the system to stabilize following a system change or event – must be accounted for in the timeout duration. In addition, there is processing delay from the time of system stabilization until the change is reported. This is especially evident in the monitoring of pressure derivatives, where subtle changes must be discerned for system diagnosis. Monitor and RTI policy changes which will reduce the diagnostic delay without compromising diagnostic accuracy are currently being investigated; recent results indicate that latencies on the order of 5 seconds can be achieved.

PITEX used approximately 1.8MB of static memory and approximately 570KB of stack memory for all of its subprocesses. Depending on the scenario, PITEX allocated between 2.5 and 3.5 MB of dynamic memory. Figure 7 gives an overall worst-case picture of total memory use. This version of the PITEX software uses the 24-bit addressing feature of PowerPC microprocessors to achieve a faster and more compact code. Employing this feature limits the maximum amount of memory accessible by the VxWorks kernel to 32MB; PITEX only uses a total of 9.5 MB, or 29.8% (including the VxWorks kernel).

With respect to CPU usage, it was found that for both the nominal and failure scenarios, the average CPU usage did not exceed 3%. However, when events are reported and a diagnosis is requested from Livingstone, the CPU usage can spike considerably. These brief spikes were as high as 99.7%. Figure 8 shows CPU usage as a function of time for the major PITEX subprocesses. In this failure mode, the series redundant helium pressurization system pressure regulators both regulate high starting at 9000 seconds. CPU usage is depicted in the final stage of captive carry when Livingstone has to perform multiple diagnoses prompted by

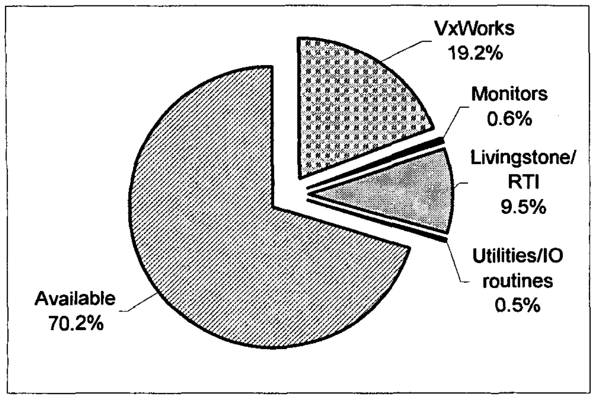


Figure 7 - Overall memory usage summary

the injected fault. As can be seen, Livingstone and the RTI are the biggest CPU consumers when a diagnosis is being performed. The spikes in the Data I/O CPU usage reflect transmission of Monitor and Livingstone findings through the telemetry channels.

If PITEX software needs to share CPU resources with other applications, it can be internally restricted to never exceed a certain percentage of CPU utilization; this feature has been successfully demonstrated on multiple scenarios at levels as low as 5%. The quality of the diagnosis was not affected by such a restriction, only its speed. Figure 9 shows the gradual increase in diagnostic delay as the CPU is restricted for six scenarios.

Finally, telemetry experiment results showed that the amount of telemetry downloaded depends to a large degree on the number of candidates reported by Livingstone throughout the mission; this dependency is illustrated in Figure 10. The system state information downloaded with each candidate was found to constitute the bulk of the telemetry data. The amount of telemetry per candidate varied somewhat, depending on the complexity of the scenario and the phase of the simulation, but, on average, reporting each candidate currently results in about 33-34 KB of additional data.

The total number of candidates downloaded during the mission should not be mistaken for the number of different components implicated as being faulty. On average, Livingstone implicated 2-3 components as being the likely reasons for a single fault and ranked them by failure probability. Many candidates were reaffirmed from one diagnostic request to another. Also, for many failure candidates, Livingstone suggested several possible times when the component might have failed. Each of these permutations is currently counted as a separate failure candidate report. Work is being done on combining several candidates into a single report and thus reducing the amount of telemetry being downloaded.

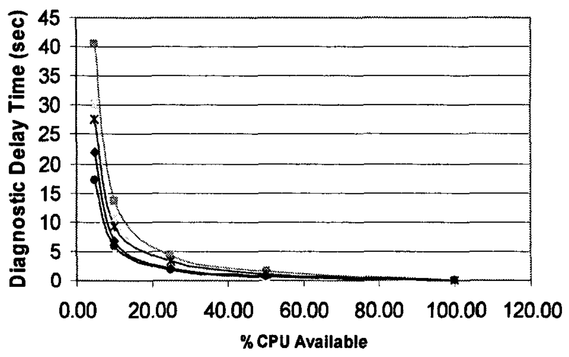


Figure 9 - Increase in diagnostic delay as a function of available CPU for six failure scenarios

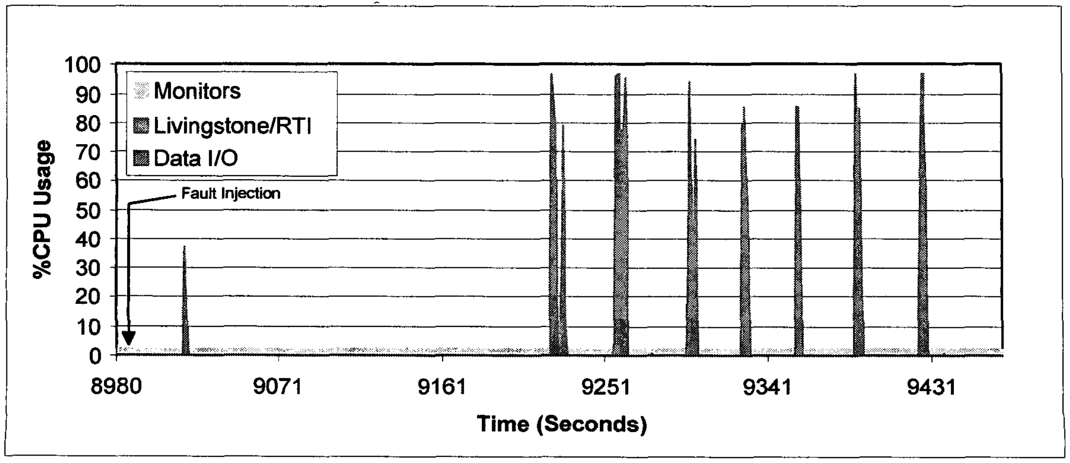


Figure 8 - CPU usage per major PITEX subprocess for double pressure regulator failure scenario

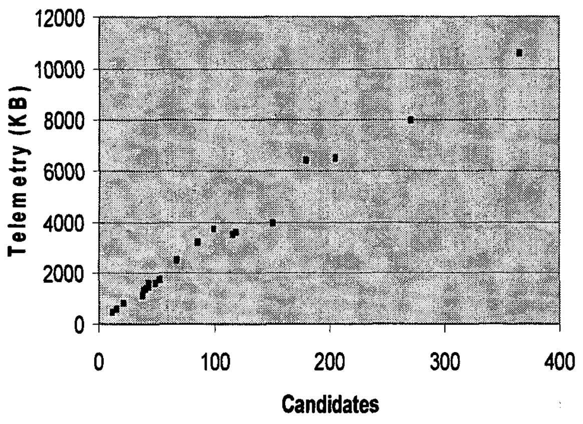


Figure 10 - Mission telemetry volume as a function of the number of candidates

CONCLUDING REMARKS

As part of the 2nd Generation Reusable Launch Vehicle Risk Reduction effort, PITEX has successfully demonstrated real-time model-based fault detection of a virtual main propulsion system. Realistic propulsion system failures involving valves, regulators, microswitches and sensors were simulated and correctly diagnosed by PITEX. The PITEX demonstration architecture included both a flight-like avionics box and a Ground Processing Unit. Specific software elements that were implemented focused on real-time system-level diagnosis. These elements were Monitors for processing raw sensor data, a Real-Time Interface for ordering and transmitting events and requesting diagnoses, and Livingstone, the diagnostic engine responsible for inferring the health of the MPS. The demonstration architecture was part of a larger more generic architecture that was designed to cover all phases of mission operation and both on-board and on-ground assessments. During testing of the software on flight-like hardware, system resources – CPU and memory – were found to be largely underutilized. This indicated that more complex applications could be handled by the PITEX diagnostic solution. The PITEX project continues to address challenges aimed at improving the speed, efficiency and timeliness of the diagnoses and is exploring other potential applications.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of all of the other current and former PITEX and NITEX team members at the NASA Kennedy Space Center, the NASA Ames Research Center and the NASA Glenn Research Center. Their contributions were instrumental in the successful completion of the work reported herein. The authors would also like to acknowledge the support and guidance of the SLI IVHM Project Office managed by the Ames Research Center.

REFERENCES

[1] P. K. Sgarlata and B. A. Winters, "X-34 Propulsion System Design," *33rd AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, July 6-9, 1997.

[2] R. H. Brown, Jr. and F. J. Darrow, Jr., "X-34 Main Propulsion System Design and Operation," *34th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, July 13-15, 1998.

[3] A. Bajwa and A. Sweet, "The Livingstone Model of a Main Propulsion System," *IEEE Aerospace Conference Proceedings*, March 2003.

[4] J. Kurien and P. Nayak, "Back to the Future for Consistency-based Trajectory Tracking," *Proceedings of 7th National Conference on Artificial Intelligence,* 2000.