The NASA Integrated Vehicle Health Management technology experiment for X-37

Mark Schwabacher*[a], Jeff Samuels[a], and Lee Brownston[b]

[a]NASA Ames Research Center, MS 269-3, Moffett Field, CA 94035
[b]QSS Group, Inc.

ABSTRACT

The NASA Integrated Vehicle Health Management (IVHM) Technology Experiment for X-37 was intended to run IVHM software on board the X-37 spacecraft. The X-37 is an unpiloted vehicle designed to orbit the Earth for up to 21 days before landing on a runway. The objectives of the experiment were to demonstrate the benefits of in-flight IVHM to the operation of a Reusable Launch Vehicle, to advance the Technology Readiness Level of this IVHM technology within a flight environment, and to demonstrate that the IVHM software could operate on the Vehicle Management Computer. The scope of the experiment was to perform real-time fault detection and isolation for X-37's electrical power system and electro-mechanical actuators. The experiment used Livingstone, a software system that performs diagnosis using a qualitative, model-based reasoning approach that searches system-wide interactions to detect and isolate failures. Two of the challenges we faced were to make this research software more efficient so that it would fit within the limited computational resources that were available to us on the X-37 spacecraft, and to modify it so that it satisfied the X-37's software safety requirements. Although the experiment is currently unfunded, the development effort resulted in major improvements in Livingstone's efficiency and safety. This paper reviews some of the details of the modeling and integration efforts, and some of the lessons that were learned.

Keywords: Diagnosis, model-based reasoning, IVHM, spacecraft, spaceplane, RLV, X-37, Livingstone

1. INTRODUCTION

Perhaps the most substantial single obstacle to the progress of space exploration is the cost of launching to, and returning from, space. The primary influence in the high costs of current systems is the operations, maintenance and infrastructure portion of the program's total life-cycle costs. Incorporation of Integrated Vehicle Health Management (IVHM) technologies into 2nd and 3rd Generation Reusable Launch Vehicles (RLVs) will result in significant program life cycle savings by improving reliability and lowering operational costs.

Life cycle costs for the next generation RLVs include safety and mission assurance. They will be dominated by both the cost of processing, operating and maintaining the vehicle and by the time required for turning the vehicle around between missions, but safety is still a critical component. Advanced IVHM technologies are critical both to the safety of the vehicle and to the cost-effective management of the vehicle.

This experiment was one of a number of flight and ground demonstrations planned by NASA to develop and mature critical IVHM technologies and to demonstrate the relevance and importance of IVHM technology to the future of the space transportation program. The eventual adoption of this technology will require a number of compelling experiments that demonstrate the ability of the technology to handle a broad class of failures and to perform robustly within an operational scenario. Furthermore, to gain acceptance the IVHM software must also demonstrate that it "plays well" in the avionics hardware and software environment of an operational vehicle without imposing a lot of additional requirements.

Figure 1 - The X-37 Spacecraft

1.1. Background

In July of 1999, NASA, the USAF, and The Boeing Company entered into a cooperative agreement to develop a new experimental space plane

* schwabac@email.arc.nasa.gov; phone 650-604-4274; http://ic.arc.nasa.gov/~schwabac/

called the X-37. The X-37 was intended to be the first of NASA's fleet of experimental reusable launch vehicle demonstrators to test future launch technologies in both the orbital and reentry phases of flight. The X-37 is an unpiloted vehicle that could be launched from the Space Shuttle's cargo bay, or from an expendable launch vehicle, and then orbit the Earth for up to 21 days before landing autonomously on a runway. The X-37 is shown in Figure 1.

There were originally 32 technology demonstrations and 8 experiments on the X-37, including the NASA IVHM experiment. The IVHM experiment was to demonstrate IVHM technology developed at NASA Ames Research Center (ARC). The IVHM experiment was led by NASA ARC working in conjunction with Boeing. A Task Agreement between NASA ARC and Boeing was signed on March 6, 2000.

The X-37 IVHM experiment incorporated the Livingstone model-based health management system[1]. The Livingstone system was initially demonstrated as part of the Remote Agent Experiment on Deep Space One (DS1)[2]. Livingstone is a model-based inference engine that reasons about system-wide interactions to detect and isolate failures. It uses a high-level qualitative model of system components in which both nominal and off-nominal modes are modeled. The Livingstone system uses this model to track commands as they are sent to components and detects discrepancies between the available observations and the predictions of the model. When an anomaly is detected, Livingstone uses advanced techniques to efficiently search the space of both single- and multiple-point possible failures to select the most likely failure that is consistent with the available observations. Once the failure is identified, Livingstone continues to monitor the system using its knowledge of the nominal and off-nominal behavior of the failed components.

Livingstone provides a number of potential advantages to 2nd and 3rd generation RLVs. Livingstone uses a high-level, qualitative model identifying components and their interaction to perform system-level health management. By using a model of the actual system, Livingstone is able to reason about complex system interactions within a real-time monitoring and control loop, rather than requiring an engineer to reason through all possible interactions and then program in the appropriate response to a pre-defined set of failures. Also, as changes are made to the hardware design, updating and verifying the model is straightforward and less labor intensive than the task of identifying changes required in explicit procedural code. The model-based level of representation streamlines the software development process and maximizes code reusability across vehicles. Finally, the use of a model facilitates the generation of an explanation or justification of the diagnosis, allowing the human operator to decide whether the diagnosis is reasonable before selecting or confirming the appropriate recovery action.

Note that Livingstone is only one of many technologies that are relevant to the broader task addressed by an IVHM system. This experiment was designed to provide only a limited demonstration of select IVHM capabilities. Specifically, this experiment focused on the real-time processing of component health information to provide fault detection, isolation, and recovery while in operation and during vehicle checkout. It was not designed to detect subtle degradations in component performance that require a maintenance response prior to the next mission (prognosis). The experiment would only identify required maintenance actions when their association with a failure could be defined *a priori* and the failure was observable within the available sensor data. The experiment was, however, a first critical step towards a more complete real-time and life-cycle-based IVHM system for enabling safe, cost-effective re-usable launch vehicle fleets.

In February 2002, due to lack of funding, effort on this experiment was suspended before the first round of integration with the vehicle. Several development cycles were never implemented. The original experiment scope is mentioned below, but this document primarily describes the final state of the integrated flight software for the experiment when it was ready for initial integration efforts.

1.2. Objectives

The goal of the X-37 IVHM flight experiment was to advance the technology readiness level (TRL) of the Livingstone reasoning system within a flight environment and to begin its transition from experiment status to inclusion in future operational vehicles.

The main objectives of the X-37 IVHM experiment were to 1) provide a limited demonstration of the benefits of in-flight IVHM to the operation of an RLV, 2) advance the TRL of Livingstone within a flight environment, and 3) demonstrate that the IVHM software could operate with the Vehicle Management Software (VMS) on the Vehicle Management Computer (VMC). To meet these objectives the flight software experiment would monitor sensor data from selected subsystems, perform real-time fault detection and isolation, and identify potential recovery actions of the X-37 vehicle during flight operations and during ground processing and checkout, while running on the X-37 VMC in shadow mode.

A key aspect of this experiment is that the IVHM software would be running in the same VMC as the flight-critical VMS. This experiment differed from other X-vehicle IVHM programs that have run the IVHM code on a separate computer, and represented an important step towards more efficient on-board computer utilization. By demonstrating the safe, integrated operation of IVHM and VMS software combined in a single software module, running on a single computer, confidence would be gained for this type of implementation on future reusable launch vehicles. Many of our challenges were tied to constraints resulting from this goal to implement IVHM on the existing flight computers.

Some of the specific functional requirements for the IVHM software experiment included:

1. The flight experiment shall be capable of identifying certain non-nominal states from sensor readings.
2. The flight experiment shall be capable of identifying sensor failures when possible from redundant sensors or consistency checks among multiple sensors.
3. When sensor failures are ruled out, the flight experiment shall be capable of declaring subsystem performance anomalies from non-nominal data. The goal shall be to isolate the failed component.
4. Diagnostic information shall be provided to the VMS for downlink to the ground.
5. The data downlinked shall be sufficient to determine the status and performance of X-37 subsystems/components as well as verification of experiment performance.

1.3. Scope

The IVHM software was originally planned to assess the health of the X-37's control surface and nosewheel steering Electro-Mechanical Actuators (EMAs) and associated Electrical Power System (EPS) status during all mission phases, with an emphasis on pre de-orbit checkouts, re-entry, and pre- and post-launch. The EMAs control the aerodynamic surfaces during re-entry and landing, and nosewheel steering during landing

Two orbital flights (of 2- and 21-day duration) and several approach and landing tests (a.k.a. drop tests) were planned. The X-37 IVHM experiment was to be part of all of these tests, along with an “IVHM day” in the 21-day orbital mission during which the potential of a real-time in-flight IVHM system was to be demonstrated by injecting “faults” into the sensor data. The initial scope of the model was intended for the drop tests with an understanding that there would be time to rescope/upscope the domain of the Livingstone model before the orbital flights.

The X-37 VMC ran the VMS software as a VxWorks task. The VMS had several functions, including managing communications with the ground and managing the IVHM software. The IVHM software also ran on the VMC as a VxWorks task. The IVHM task received inputs (sensor values and commands) from the VMS task, and provided its outputs (diagnoses) to the VMS task to be downlinked to the ground.

To ensure vehicle safety, the experiment would be constrained to “monitor only” (“shadow mode”) during flight operations due to technology readiness levels and aggressive cost and schedule goals. While mission operators and processing personnel could look at the results of the experiment during a mission, the operating procedures for the X-37 would not depend upon the results of the IVHM experiment. Flight managers would not rely upon the information from the IVHM experiment during any mission operations, but the outputs could be used to assist in operational decisions if so desired.

There were six major task areas that NASA performed to get Livingstone ready to fly on X-37:

1 Port the Livingstone inference engine from LISP to C++ and meet Boeing safety requirements.
2 Develop the interface between Livingstone and the X-37 VMS
3 Develop monitors to be used by the Livingstone model (see Section 2.2)
4 Develop a Livingstone model of the X-37 vehicle subsystems
5 Integrate the model, the monitors, the interface code, and the inference engine, and perform testing of the resulting system.
6 Develop Ground Station software

1.4. Overview

The basic functionality of the software relies on vehicle sensor data and commands, which are provided to the IVHM software by the VMS. Changes in sensor state are detected by feature extraction monitors, the outputs of which are used by Livingstone. The vehicle status, nominal or otherwise, is regularly provided to the VMS for downlink. The experiment status is likewise provided for downlink.

Figure 2 shows the architecture of the IVHM experiment. The VMS, written by Boeing, and the IVHM software would run on the same VMC. Among its other functions, the VMS commands the power system, acquires subsystem status and sensor information, monitors commands issued by the Flight Management Computer (FMC), and stores vehicle state information in a shared memory area. The IVHM retrieves the vehicle state data and commands from shared memory. Monitors translate the real-valued sensor data into a discrete representation to be used by Livingstone. These discrete values are then fed into the vehicle model. The vehicle command stream is also made available to the IVHM software in order to identify divergent behavior. Livingstone generates the diagnoses. The IVHM software will never directly command any vehicle subsystems and under no circumstances would its recommendations be acted upon automatically by the vehicle. Rather, the IVHM outputs are placed in the IVHM area of shared memory for insertion into the telemetry stream and relay to the ground station. The VMS then telemeters this information to the ground, where ground software processes and displays the information.

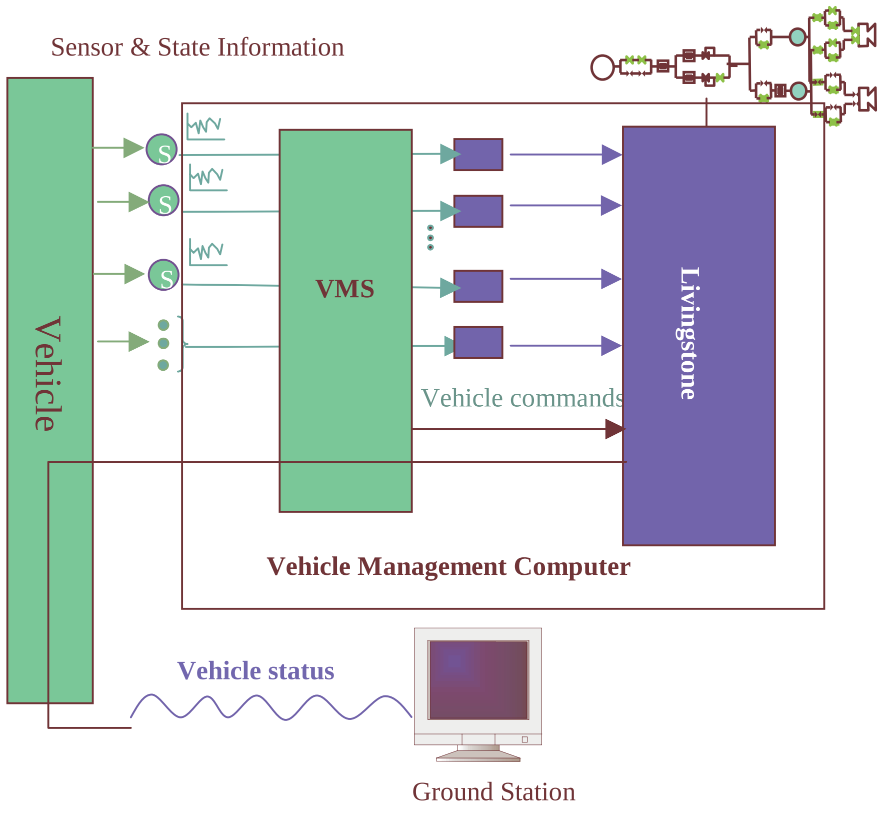

Figure 2 — X-37 Livingstone Overview

1.5. Challenges

Running Livingstone on the VMC presented some special challenges. First, the resources available to the experiment (CPU, memory, and telemetry bandwidth) were very limited. Second, the VxWorks 5.4 operating system does not provide any memory protection, so a bug in our software that caused it to write into the VMS task's memory space could crash the VMS and thereby risk loss of the vehicle. Because of this risk, Boeing required our code to meet certain safety standards (such as no dynamic memory allocation and no pointer arithmetic), and to undergo very thorough testing.

2. SOFTWARE COMPONENTS

2.1. Livingstone

Livingstone is a software package that uses a search process to diagnose problems in a complex system that has been described using a declarative model[1]. Livingstone was developed at NASA ARC. A previous version of Livingstone, which was written in LISP, was successfully flown on board the Deep Space 1 spacecraft[2]. The current version of Livingstone (L2) is written in C++. In order to fly L2 on X-37, the X-37 IVHM team completed the following modifications to L2:

1. L2 was ported to VxWorks
2. L2 was modified for flight code to satisfy the following X-37 safety standards, *e.g.*:
   - No dynamic memory allocation from the heap
   - No recursion
3. L2 was interfaced with the VMS
4. We implemented routines to allow L2 to use a binary representation of the model, to avoid the need to fly the XML parser used in the current implementation

2.2. Monitors

Because L2 is able to take only discretized values as inputs, it is necessary to translate all sensor values into discretized values. This is the task of the command and sensor monitors, which were written in C. Command monitors are functions that take as input the commands provided by the VMS (*e.g.*, 0 or 1), and translate these commands into the format needed by L2 (such as on or off). Sensor monitors are functions that convert engineering-unit sensor values into discrete values. There are four types of sensor monitors, corresponding to different types of sensors in the X-37 subsystems covered:

- TranslateOneBit monitors interpret on-off sensor values. The monitors echo the on-off value.
- TranslateSSPCStatus monitors interpret sensors with three status bits. The monitors will return the bit pattern passed from these sensors.
- Bin/threshold monitors interpret floating-point sensors that need to be placed in a "bin" where a range of values corresponds to a bin. For example, a temperature value can be "low", "nominal", or "high" according to whether its value lies in the different ranges corresponding to these bins.
- Compare monitors interpret floating-point position sensors for the aerodynamic control surfaces. The desired values that the monitor should return are "moving" and "not moving". Thus, the compare function uses the difference between successive sensor values to determine whether its output should be "moving" or "not moving"

Because the signal from the sensors on the spacecraft contains noise, some filtering is required to prevent L2 from interpreting signal noise as a sensor indicating a failure. Both persistence and confidence filtering were implemented in the monitors.

Persistence is related to the transition of a sensor signal from one value to another. It is the determination of when the new value should be reported by the monitors. In other words, it is the determination of when we are certain that the new value is correct and not merely caused by noise. The implementation is quite simple. We keep track of a counter for each possible output value. We also keep track of the value that was previously reported by the monitors. Also, a persistence level is set as a parameter. The higher this parameter is, the more counts are required to switch the discrete value reported by the monitor. Each counter can have a value ranging from 0 as a minimum to the persistence level as a maximum. Whenever a discrete value, calculated from the engineering value, is encountered that is not the same as the previous value, its counter is increased by 1. All the counters for all the other values are decreased by 1. If a value is not within the allowable range, all the counters for the values will decrease by 1. If one of these bin counters reaches the persistence level, then the monitor will report the new value.

Confidence is the determination of whether the monitors should report "unknown" when a sensor output is transitioning from one value to another, but cannot be determined with certainty as belonging to either value. In this implementation, confidence is determined independently from persistence. There are 3 states of confidence:

- Full confidence - equivalent to reporting the current bin consecutively *n* times, where *n* is the persistence level.
- Partial confidence - monitor reports previous value, but if next output from the monitors is not the current output, the monitors will report "unknown".
- No confidence - the monitor reports "unknown"

Also, three parameters are set: 1) Max number of consecutive "not current discrete" values before switching to partial confidence, 2) Number of consecutive current values needed to change from a partial confidence state to full confidence, and 3) Number of consecutive current values needed to change from a no confidence state to a partial confidence state. Thus there are 2 counters for each discrete value, one for consecutive "not current values" for switching to partial confidence, and another for restoration of partial and full confidences.

Monitors are “tunable” in that the transition ranges are set by parameters. Persistence and confidence filters are also “tunable” as described above. When tuning the noise filtering in the monitors, there is a tradeoff between false positives and false negatives. If there is not enough noise filtering, there is the risk that L2 will diagnose a failure when there has not in fact been a failure in the vehicle. If there is too much noise filtering, there is the risk that a real failure will be missed because it is assumed to be noise.

2.3. Models

L2 makes use of a declarative model of the system being diagnosed. ARC completed an initial model of the X-37’s EPS and EMAs. This model was compiled into a binary file, which was to be provided to Boeing for inclusion in the VMC’s Flash RAM.

The initial scope of the model was intended for the drop tests with an understanding that there would be time to rescope/upscope the domain of the L2 model before the orbital flights. For example the low-voltage battery and charging system, or the individual cells of the high-voltage battery, or system redundancy might have been modeled. This paper describes the L2 model for the X-37 drop tests.

The initial L2 model for the X-37 consists of a single thread (no redundancy) from the high-voltage batteries to the EMAs, although the X-37 does have redundant power sources and buses. Six of the nine actuators are modeled (the nosewheel steering and thrust vector control actuators are excluded). Systems modeled include: high-voltage battery, high-voltage power relay, SSPCs (solid state power controllers), EMA controllers, and EMA motors and brakes.

Figure 3 shows the top-level schematic of the model. Figures 4 through 7 highlight sub-modules within the top-level module.

The model consists of four main module types: High-voltage battery, Power Control and Distribution Unit (PCDU), actuator controller, and actuator. The first two types constitute the electrical power system while the last two constitute the flight actuation subsystem. In the current model, there are 9 distinct modules, 18 distinct components, and 86 component instances as depicted in the following figures.

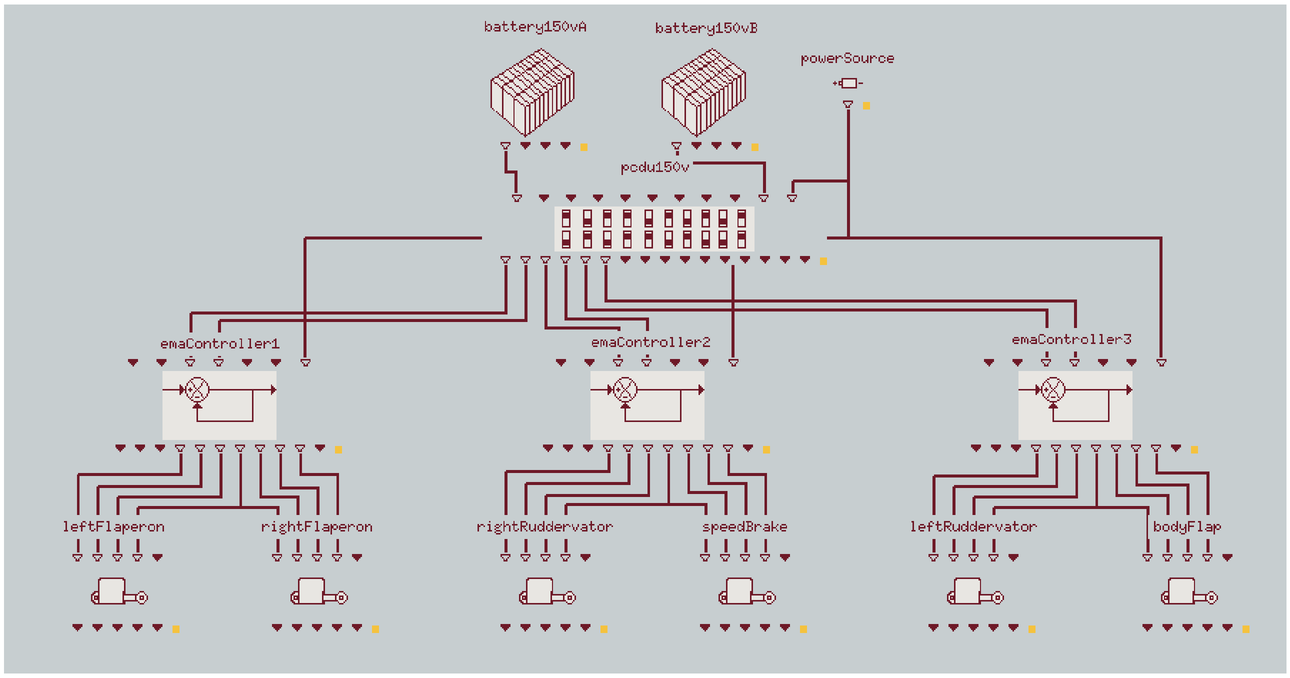

Figure 3 — X-37 Livingstone model.

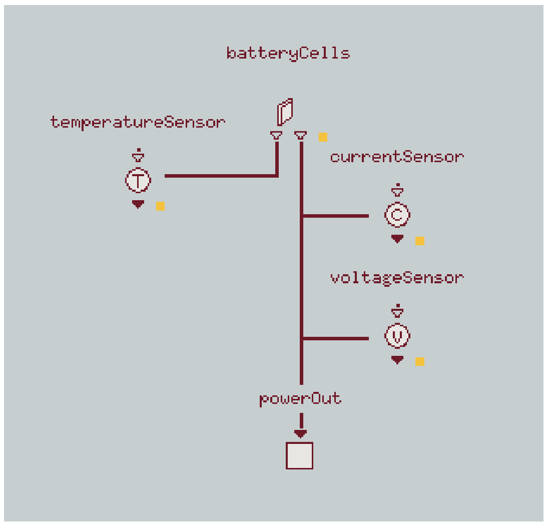

Figure 4 — Battery module.

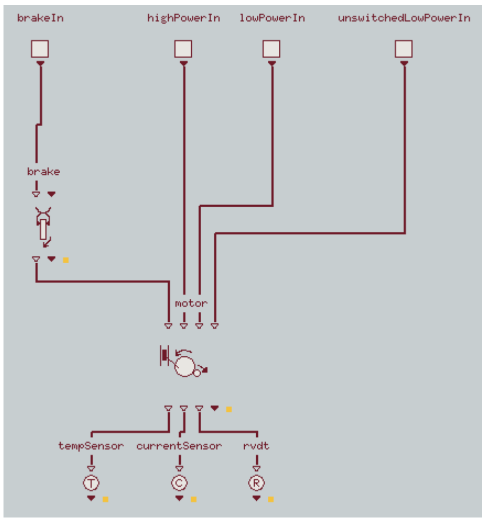

Figure 5 — Actuator module.

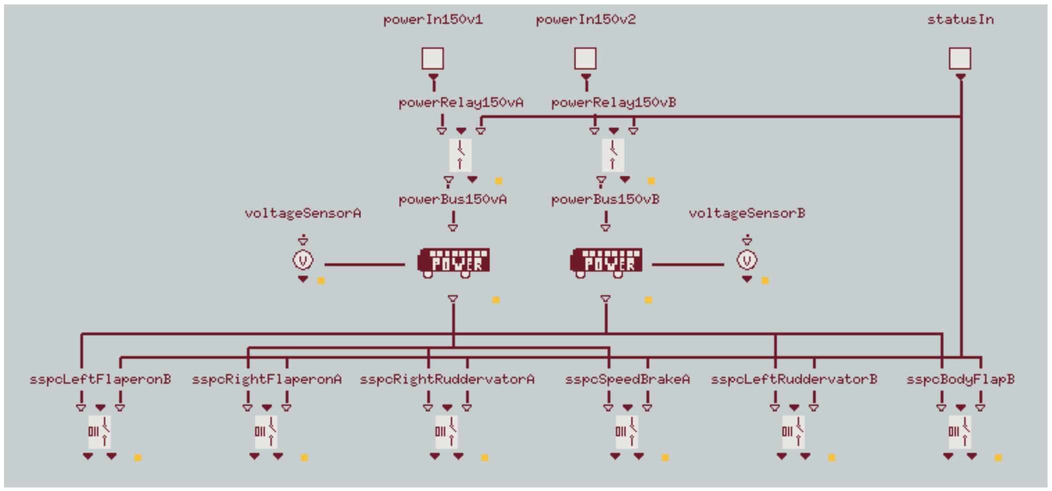

Figure 6 — PCDU module.

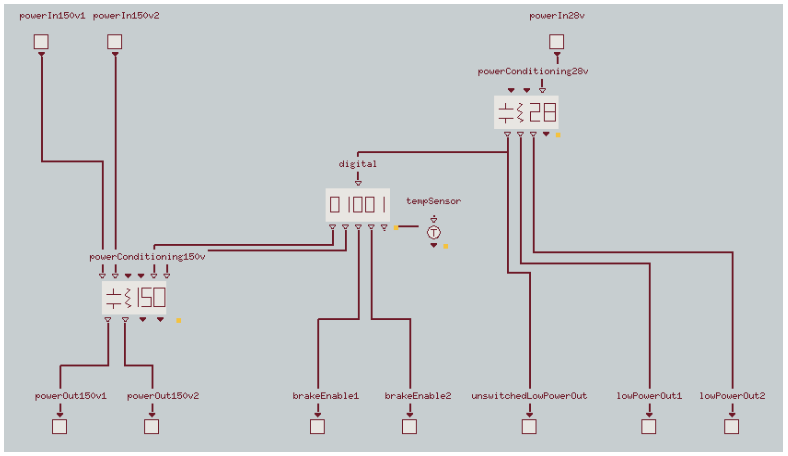

Figure 7 — EMA Controller module.

3. INTEGRATION

3.1. Design constraints

In addition to the modifications made to L2 for vehicle safety requirements, there were several other constraints on the experiment software. These included memory, CPU allocation, and telemetry bandwidth. These issues, along with L2 performance, have tremendous implications in terms of 1) L2's ability to perform real-time diagnoses (if it were to generate recovery options for execution), 2) L2's ability to provide operators timely guidance on failures and appropriate recovery, and 3) the ability of experimenters to validate L2 diagnoses in real time and post-test.

3.1.1. Memory constraints

The IVHM software was required to fit within a very limited amount of DRAM and Flash RAM on the X-37 vehicle. The size of the code and the amount of space it used for data had to be reduced substantially in order to fit within the requirements.

3.1.2. CPU constraints

The IVHM software was developed to run as a separate VxWorks task. The IVHM task was allowed a very small, fixed amount of CPU time on a fixed cycle. This constraint introduced the unknown element of how L2 performance (dependent on the model size and the actual sensor/command stream) would lag real time. Greater lag also requires a larger input queue to store the sensor/command stream as described later.

3.1.3. Telemetry constraints

The X-37 will have telemetry on two bands: one low-speed, and one high-speed. The low-speed telemetry will use well-known, redundant technology, which is expected to be reliable and have high coverage, but has a very low expected throughput. The high-speed telemetry is an experiment, with no redundancy, but with a very high expected throughput. The low-speed telemetry channel is redundant in order to provide highly reliable and nearly continuous coverage during the orbital missions. This channel is therefore reserved for high-priority or mission-critical data, so Boeing constrained the IVHM task to a very small portion of the bandwidth on it. NASA was also to get an unspecified amount of bandwidth on the high-speed telemetry with intermittent coverage. These limitations presented the IVHM team with a significant challenge. The interface between L2 and the VMS had to balance the desire to get information to the ground (operational and experiment status) with the limited RAM available for output queues.

3.2. Communication

All experiment communications (receiving the sensor and command streams, and telemetering output) is done via the VMS. The IVHM software communicates with the VMS using three queues that are stored in shared memory. At a fixed frequency, the VMS uses the monitors to filter the relevant sensor values and process the relevant commands, and then places the filtered sensor values and processed commands into the input queue. Next, the IVHM software removes filtered sensor values and commands from the input queue until a diagnosis or state update is needed. Each time L2 completes a diagnosis, the diagnosis, represented as the incremental change in the state vector, is placed in the two output queues (one for each telemetry band). This also creates the challenges of deciding what goes into each queue and synchronizing these outputs on the ground. On each cycle through its main loop, the VMS removes as many items from the output queues as there is bandwidth available for downlink. NASA ARC developed a function to be called by the VMS at a fixed frequency that, in turn, calls the monitors, adds the monitor outputs to the input queue, allows L2 to run for a fixed amount of time, retrieves L2's outputs from the output queues, and returns these outputs to the VMS.

3.3. Integration architecture

The IVHM software uses three queues: an input queue, a low-speed output queue, and a high-speed output queue. All three are first-in, first-out (FIFO). The VMS places monitor outputs (filtered sensor values and commands) into the input queue at a fixed rate. Thus a single input queue holds all of the monitor outputs that are input to L2, in chronological order. The IVHM task removes these items from the input queue whenever it is ready for them (in between diagnoses). The IVHM software places L2's outputs (diagnoses) into the output queues. The plan was for it to place a minimal description of its diagnoses into the low-speed output queue, and a more complete description of its diagnoses, and how it arrived at them, into the high-speed output queue. The VMS removes items from the two output queues at a fixed rate and downlinks them on the appropriate bands. We only implemented the minimal description (low-speed output queue). The integration architecture consists of the following components, shown in Figure 8:

1. **Input queue writer (IQW)**. The VMS calls the IQW function, passing it engineering-unit sensor values and commands. IQW calls the monitors, which transform the engineering-unit sensor values into "monitor outputs," which are filtered sensor values that have been discretized. It also calls monitors to translate the

commands into the format needed by L2. IQW then places the monitor outputs into the input queue (only commands and discretized sensor values that have changed).

2. **Input queue reader (IQR).** This function removes monitor outputs from the input queue, passes them to L2, and calls L2 as necessary to perform diagnoses (see diagnosis timing policy, below).
3. **Monitors.** The functions that take as input the engineering unit sensor values and commands provided by the VMS, and output discrete values that can be used by L2. (M1, M2, …, Mn in Figure 8.)
4. **L2.** The inference engine that performs a diagnosis.
5. **Output queue writer (OQW).** The function called by L2 to put a diagnosis and related information into the two output queues.
6. **Output queue reader (OQR).** The function that the VMS would use to remove diagnoses from the two output queues so that they can be downlinked.

If any of the queues becomes full, the experiment is terminated.

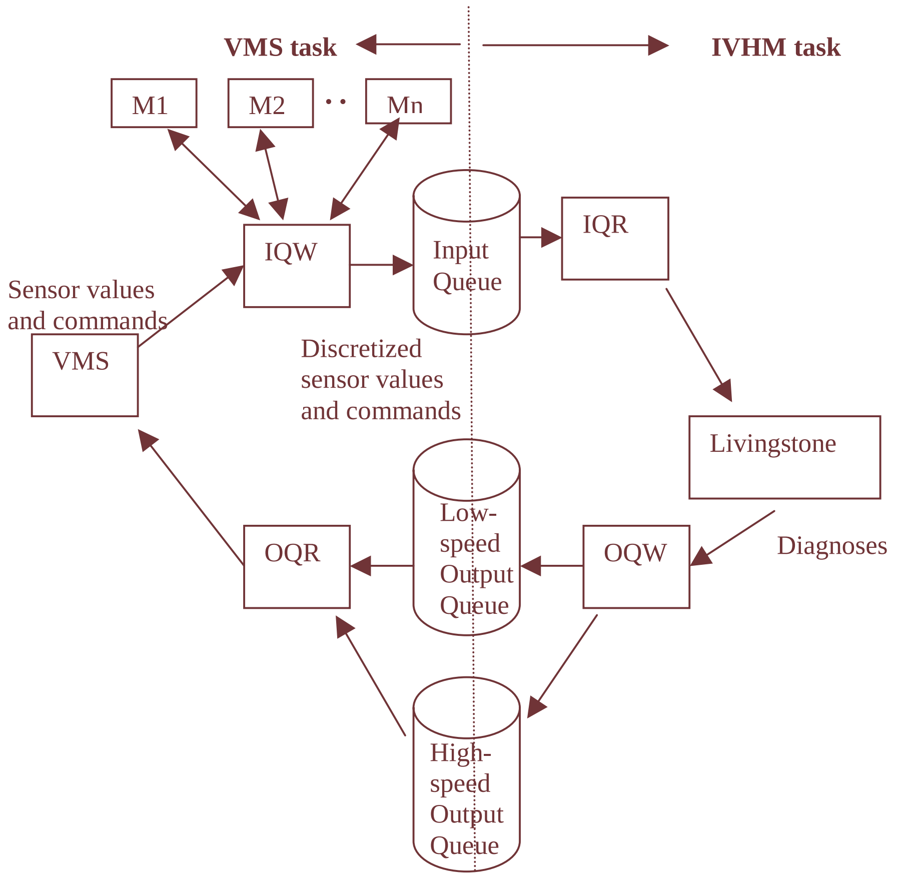

Figure 8 - IVHM Experiment Software Architecture

3.4. Diagnosis timing policy

The policy engine is implemented within the IQW, which is within the VMS task. The policy engine decides when a diagnosis should be done, and then inserts a call to “find candidates” into the input queue. When the IQR removes this “find candidates” from the input queue, it instructs L2 to perform a diagnosis.

The policy engine incorporates “tunable” parameters into the interface. When a command is received, there is generally a small time delay before the effects of that command can be seen in the sensor values. If L2 performs a diagnosis immediately after receiving a command, then it may produce an incorrect failure diagnosis, since the effect of the command may not yet have appeared in the sensor stream. One solution to this problem is to associate with each command a list of sensor monitors that could be affected, and the amount of time for each effect to appear. After a command, L2 would ignore each specified sensor for the specified period of time. We selected a simpler solution, in which there is a global time delay for all commands. After a command is issued, the system waits for the specified amount of time, and then does a diagnosis, unless another command is received during the delay. So, in effect, after the first command is issued, the system waits until there is a period of the specified length without any commands, and then does a diagnosis. In addition, if there is an unexpected observation (one that does not follow a command), the system does a diagnosis after a delay (assuming that there is no command during that delay).

In selecting the timeout after an unexpected observation, there is a tradeoff between alerting operators to the problem as soon as possible so that they can act on it, and waiting until enough data arrives to produce the correct diagnosis. We tried to implement a system that will obtain the correct diagnosis. It might be useful to do both: alert the operators of a failure as soon as the first evidence of the failure arrives, and then provide the diagnosis of the failure as soon as enough data has been received to diagnose it. Of course, there are implications here in terms of implementing a control system with L2 “in the loop”.

4. TESTING

Testing of the initial version of the software fell into safety, performance, accuracy, stability, and operational areas. We verified that the safety requirements were met by examining the source code, either manually or using tools that search the source code for prohibited items. We tested memory usage and verified statement coverage using CodeTEST.

Stability testing included nominal scenario and random failure testing on a PowerPC at NASA ARC similar to hardware on the X-37. These tests confirm code stability and memory use with and without failures. We successfully simulated a nominal (no failures) 21-day orbital mission in 21 days of continuous execution on the NASA ARC processor without exceeding memory allowances. Stability testing was also conducted by providing random inputs (Monte Carlo testing) to the integrated software package. These random inputs produced random monitor outputs that correspond to random failures. The frequency of failures generated in this manner is unrealistic on a vehicle. Using the Monte Carlo approach, we simulated eleven 21-day missions in faster-than-real time. The Monte Carlo tests did not allow us to verify accuracy, since we do not know what the correct diagnosis is for any of the resulting random inputs. They simply allowed us to verify that the code does not crash or exceed its memory allocation, given a very wide range of different inputs.

Accuracy testing was performed using a set of 28 hand-coded scenarios. No simulation data was available to test the accuracy of the L2 model and monitors in diagnosing faults in the EMAs and associated EPS. Instead, scenarios were constructed based on expected sequences of commands and observations for nominal and failure operating conditions. The system produced what we believe to be the correct diagnosis in every case. It should be emphasized that Boeing engineers did not confirm the applicability of the scenario files or verify the correctness of the diagnoses.

This version of the IVHM software is not ready for flight as of the date of this report, because some of the software deliverable acceptance tests either did not pass or were not tested. The next step in testing our software, had this effort continued, would have been a functional test conducted with Boeing to see if the IVHM software could have been compiled and executed on their system.

5. CONCLUSION

5.1. Accomplishments

Although the X-37 IVHM experiment was suspended, a number of accomplishments were made towards the technical goals. First, Livingstone software was ported from LISP to C++ under VxWorks. During this process, Boeing memory and CPU usage requirements were satisfied. Specifically, DRAM usage was reduced 85%, Flash RAM usage was reduced 84%, and speed was increased about 1600%. Most of Boeing’s safety requirements were also met. For example, all dynamic memory allocation from the heap was removed from the code.

The software team completed the design and development of additional flight software components required for the integrated IVHM software package (an interface to Boeing’s VMS and a testing environment). This included completion of version 1 (for atmospheric flight tests) of the X-37 subsystem models that included a single string of the EMAs and a partial EPS. The team had originally planned an expanded second version of the model (for orbital flight tests) before the experiment was suspended. In addition, monitor routines were developed which act as an interface between vehicle data (in engineering units) and L2.

It was originally planned that three ARC internal and three Boeing versions of the integrated flight code would be developed (for integration with the vehicle, atmospheric flights, and orbital flights). The first ARC internal integrated IVHM system was completed in November 2001. This system integrated the subsystem model, monitors, test harness, I/O queues, and VMS interface code with the modified flight version of L2. Testing through February 2002 successfully demonstrated the stability of the software on a ground-based system similar to the flight computer/software environment. The next phase of development would have been working with Boeing to integrate and test this version with their hardware/software on the ground. However, X-37 delays and subsequent cancellation of the experiment prevented this integration.

No further work on this experiment has been funded. An accepted approach to integration was successfully developed that would probably have led to safe integration with vehicle flight software. The work remaining if this code were to be flight-tested includes: 1) In-flight IVHM error handling and L2 initialization/restart capability, 2) Successful integration with flight hardware and software, and 3) Integration with ground station software.

5.2. Limitations and future work

A number of challenges and questions arose during development of our software. We began to think of solutions for some of these items before this task was terminated. We did not have the opportunity to address these issues or even make serious recommendations. Many of them currently exist only as questions.

Much of the work in creating a model is acquiring the knowledge of the systems to be modeled. What are the interfaces to the system? What are the observable commands and sensor readings? What is the normal operation and how do faults of the system manifest themselves in the observable parameters? A significant challenge of the modeling effort for X-37 was having only limited interaction with the domain experts. Assumptions had to be made about the system components and operation. Another difficulty was the dynamic nature of the design. As the systems change, so too must the model and monitors.

One should have a good understanding of the systems being modeled before applying Livingstone to diagnose them. Obviously, model-based reasoning tools can diagnose only what has been modeled. Unanticipated system behavior that has not been modeled will most likely lead to a diagnosis that is highly unlikely or misleading, or an “unknown failure” which is not very useful. It is important to have simulated data in order to debug the model and monitors. Unfortunately, very little data was available for debugging purposes. The model and monitors would undoubtedly have to be modified after running through simulated data scenarios and after further interaction with Boeing engineers.

Modeling was made more difficult by the high frequency response of some X-37 subsystem components, interaction of these subsystems with embedded controllers and fault detection systems, and redundancy in the subsystems.

Choosing the domain of the model is important for a successful demonstration of Livingstone. Livingstone is a discrete, qualitative model-based reasoning engine. Continuous-valued systems must be discretized into qualitative bins. Setting the number and threshold of the bins must be accomplished by examining system-wide interactions, not just local behavior. While this can be done for many cases, the problem becomes more difficult when the expected parameter values depend upon the mission phase or upon the duty cycle of the device being modeled. A goal of model-based programming is to reduce development time and costs by building a library of components that may be plugged into other models. In order to accomplish this, the model must not have inherent dependencies on the system being modeled (function in structure). Currently, defining a model without function in structure is difficult due to the interaction of the bin definitions, component failure modes, and system-wide behavior. Also, systems with high-frequency feedback loop control present challenges to a discrete model. They must be abstracted to a “black-box” level to determine if they are functioning properly. For systems that have on-board health diagnostics, decisions have to be made about complementing and enhancing the current fault detection with Livingstone or demonstrating that Livingstone can replace the on-board system. The first approach was taken for X-37. In our model, the general rule of thumb was that when independent measurements could not confirm or refute the status provided by a device, then the status (and therefore the device) was treated as infallible.

Successful integration requires experienced modelers and close cooperation with vehicle hardware and software developers, or extensive training of vehicle developers. This, in itself, represents a challenge to acceptance of this technology in future vehicles. Presently, Livingstone experts, rather than vehicle, system, subsystem, and component level engineers, are required to build models. But they, in turn, require the cooperation of the domain experts. Darwiche[3] addresses this problem.

We implemented a simple policy engine, which uses a single global timeout value for commands, and another global timeout value for unexpected observations. We tuned these parameters to get the system to work correctly for all of our scenarios. It is not clear that the values we chose would work correctly with real flight data. Before the system could be flown, it would have to be tested with real flight data. The timeout values might have to be further tuned. It is also possible that we would discover that the simple approach we took to timing would not be sufficient, and that we would have to implement a more sophisticated approach, such as having a different timeout for each command and for each observation.

Our simple policy engine cannot handle interleaved commands – that is, the case where a second command is issued before the results of the first command can be observed. One way to extend it to handle certain interleaved commands would be to have it ignore (for a time period) only the specific observations that should be affected by a particular command.

In the current system, there is no redundancy in our telemetry. Depending on the reliability of the telemetry channel, we might have needed to implement a system that had redundancy in the telemetry. For example, we could periodically downlink Livingstone's full state, in addition to downlinking every incremental update to Livingstone's state.

We recommend the development of a more formal approach to developing the monitors and the policy engine. Currently, these parts of the system are developed by writing C code from scratch for each new vehicle that is modeled.

Livingstone must be tuned differently for each application. Tunable components include the policy engine (interface timing), Livingstone operational parameters (history length, number of candidates, accuracy *vs*. speed/size, *etc*.), and monitors (data smoothing, persistence, *etc*.). Tuning requires experience, and perhaps art, on the part of integrators. We recommend the development of a more formal approach to "tuning" the various parameters of the system. Livingstone tuning directly affects the speed of diagnoses and therefore the memory required for implementation.

One difficulty we encountered is that the X-37 vehicle behaves very differently during different mission phases (such as on-orbit, reentry, and landing), but Livingstone uses the same model and monitors for every phase. It would be useful if Livingstone could be aware of the mission phase and use different monitor parameters, different monitors, or even different models during different mission phases.

Our experience with X-37 has demonstrated the need for a system that can perform hybrid discrete/continuous diagnosis. Some parts of the X-37 systems that we modeled are inherently discrete, such as the switches in the EPS. Some parts are inherently continuous, such as the voltage in the batteries and the position of the EMAs. In some cases, such as battery voltage, it makes sense to use monitors to discretize the continuous values. But in other cases, such as EMA position, it is not practical to discretize the values. We found that the only way we could diagnose the functioning of the EMAs in a Livingstone-based system would be to effectively perform the diagnosis within the monitors. We instead chose to rely on the X-37's sensors to tell us whether or not the EMAs were working. We believe that the portion of the X-37 that we selected to model in this experiment could be better modeled in a hybrid diagnosis system.

Finally, there are a number of issues in successfully integrating flight data with a ground station. These include issues of how ground station software handles telemetry noise and dropouts, synchronization of different telemetry bandwidths, synchronization of flight and ground instances of Livingstone (desired for certain additional capabilities this would offer), not to mention the effort required to fully develop a user interface that supports real time alerts/status and archival data for reviewing past outputs.

6. ACKNOWLEDGEMENTS

We would like to thank all of the members of the X-37 IVHM Experiment team, including Mina Cappuccio, Scott Christa, Scott Poll, Jeremy Ou, Jim Kurien, Benoit Hudson, Chris Plaunt, Dan Clancy, and Erv Baumann. This experiment was funded by the X-37 Project Office at NASA Marshall Space Flight Center, and by the Design for Safety Program at NASA ARC.

7. REFERENCES

1. Brian C. Williams and P. Pandurang Nayak, "A model-based approach to reactive self-configuring systems," *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 971-978, MIT Press, 1996.
2. Nicola Muscettola, P. Pandurang Nayak, Barney Pell and Brian C. Williams, "Remote Agent: to boldly go where no AI system has gone before," *Artificial Intelligence* **103**(1-2), pp. 5-47, 1998.
3. Adnan Darwiche. "Model-based diagnosis under real-world constraints." *AI Magazine* **21**(2), pp. 57-73, 2000.