

THE MARS SCIENCE LABORATORY ENTRY, DESCENT, AND LANDING FLIGHT SOFTWARE[†]

Kim P. Gostelow[‡]

This paper describes the design, development, and testing of the EDL program from the perspective of the software engineer. We briefly cover the overall MSL flight software organization, and then the organization of EDL itself. We discuss the timeline, the structure of the GNC code (but not the algorithms as they are covered elsewhere in this conference) and the command and telemetry interfaces. Finally, we cover testing and the influence that testability had on the EDL flight software design.

INTRODUCTION

Edl is an autonomous control program for the Entry, Descent, and Landing phase of the Mars Science Laboratory (MSL) mission. The design of the Edl program inherits many characteristics from its predecessor mission, the Mars Exploration Rovers (MER), but was nevertheless a new development. This paper describes the design, development, and testing of parts of the Edl program from the perspective of the software engineer. Guidance, Navigation, and Control (GNC) algorithms are covered elsewhere^{1,2}.

MSL COMPUTING HARDWARE AND SOFTWARE

The MSL avionics computing system comprises two RAD750 processors with 256 MB of RAM each, and various flavors of non-volatile memory of several gigabytes. Some time before Edl, one of the two computers is designated prime and the other backup. There is no off-loading of work onto the backup computer; the backup exists solely in case the prime machine fails. The basic design has the software on both machines the same, but the prime machine runs the “prime” software, while the backup runs a different set of functions. The backup machine does useful work only if the prime fails, and only if we are in a certain sub-phase of Edl. In an alternative formulation, called Second Chance, the backup machine runs different software than the prime machine. Some days before Edl, it was decided to allow Second Chance to be used, if needed. Second Chance is described briefly later.

The RAD750 connects to two 1553 busses, one running at 8 hz and the other at 64 hz. The busses connect to a large number of peripheral devices including radar, power, telecom, pyro, science instruments, and others. The inertial measurement units (imu) had a special connection to the processors so the 200 hz measurements could be processed with as little delay as possible.

[†] © 2013 California Institute of Technology. Government sponsorship acknowledged.

[‡] Principle Software Engineer, Jet Propulsion Laboratory – California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91009.

Virtually all other device communication was over the 1553. Many devices were redundant as were their cross-coupled controllers/drivers. However, due to switching delay and the need for concurrent data paths and computation, there are few opportunities in the Edl system to utilize the redundancy in real time in case of problems.

The software is written in ANSI C and runs under the VxWorks real-time operating system. Lines of code is the usual measure given for the size of a program, but it can be difficult to gauge just what a line is, and which software you count. There are approximately 1 million lines of handwritten C code in the MSL flight software, and an additional 2.5 million lines were produced by an autocode system. The autocode is generated from domain-specific languages including general finite-state machine, telemetry, commands, parameter handling, and other areas. The inputs to the autocode generators are specifications, coded in XML by the programmer, of what is needed. The generators, written in python, read XML and C “.h” files, and produce C code as output. Some of the domain-specific code was also used to generate information for the ground telemetry and real-time display system.

The MSL flight software is partitioned into some 150 modules, each module containing zero or more (usually one) threads. The architecture of the flight software is communicating threads (or tasks in VxWorks parlance) all running in a single address space. Communication among threads is by messages (with some exceptions). Interrupt handlers convert interrupts to messages as soon as possible in the interrupt decode process. A clock tick is also a message sent to those threads that wish to receive it. The 150 threads include all device controllers, and all other mission activity for all of cruise, Edl, and landed spacecraft configurations.

THE FLIGHT SOFTWARE PRINCIPLES

The MSL Flight Software Principles were written early in the project. These principles came from approaches and lessons learned from Mars Pathfinder and the Mars Exploration Rover projects. The Edl modules are indistinguishable from any other module in the system and receive no special treatment in any way. Thus these Principles apply to Edl and were important to its development. These principles are:

1. A module communicates with another module through messages containing only data – no pointers may appear in a message.
2. Each task executes an event loop that processes arriving messages. A task waits only on message arrival and at only one point in the code.
3. There is a single manager or point of contact for any given decision.
4. Message sending is by void return functions (and not functions that return a communication success or failure indicator).
5. Wherever possible, memory is allocated statically, including stack space. The remaining area is dynamic memory, which is allocated only during initialization, before any tasks have started.
6. Task priorities are assigned according to Deadline-Monotonic scheduling theory. All messages for a given task should result in approximately the same computation time.
7. A module is a collection of software with high cohesion, low coupling, and singular purpose. It is the responsibility of, and is owned by, a single developer.

We would like to point out some implications of the principles. MSL modules might be described as behavior-oriented; this would be especially the case for the higher-level modules, such as Edl. A module contains all state and functions needed to carry out some aspect of flight software, and is the basic unit of software organization, personnel assignment, and unit test.

Threads communicate by message rather than shared memory. That is, the software is event-driven. This is important for isolation between modules. The expectation is that modules based on messages, rather than shared memory, have a higher probability of behaving as expected after integration with other modules.

It is important that a thread wait for a message at only one point in the code. Generally, a finite-state machine is used to determine what to do with any given message, but by waiting at only one place in the code, a module will have a handler for any message at any time, including messages that may violate some protocol, even if it means throwing the message away. At least the message will be handled, perhaps with a warning message to developers.

Module initialization and messaging is standardized in MSL. Within a module, a developer was pretty much free to write the code as he pleased. Some modules have a layered, function-oriented structure, while others are more object-oriented (though in C). Some modules have no thread, in which case we call them libraries.

THE EDL FLIGHT SOFTWARE

The Edl flight software comprises four principal modules:

- EdlMain with the Edl timeline and the single Edl main thread
- EdlGnc is a library containing the vast majority of GNC code that runs during Edl; the code in this library is run by the thread in EdlMain.
- EdlComm has a single thread and is responsible for recording data and sending a portion of it, in real time, to the ground
- EdlBg is the background batched radar measurement thread, and secondarily handles parameter updates for all of Edl.

Figure 1 shows the relationships among the Edl modules as well as some of the modules external to Edl.

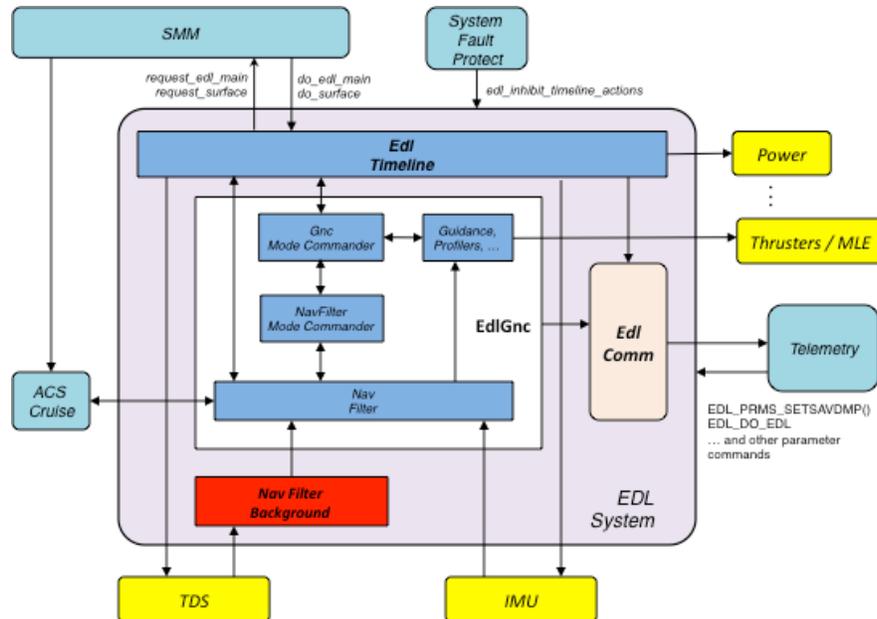


Figure 1. The Edl main thread runs both the Edl timeline and EdlGnc code.

The following sub-sections detail each of the modules in the Edl system, and how Edl works overall.

Edl and EdlGnc Modules

The Edl module has one thread, which synchronously runs the timeline and the code in the EdlGnc module. That is, the EdlGnc module has no thread of its own, and thus is a library of functions that perform GNC. Running the Edl timeline and EdlGnc under a single thread simplifies the interactions.

The timeline and GNC cooperate. The timeline is concerned with telecom settings, powering of devices, and most significantly with changes in spacecraft configuration. For example, the spacecraft transitions from a complete spacecraft with cruise stage, descent stage, and rover all connected, to the entry configuration where the cruise stage is jettisoned. Such configuration changes are orchestrated by the timeline by executing timepoints that perform actions at an appointed time. For example, cruise stage jettison is not a single timepoint action, but is a process that requires a number of timepoint actions such as venting the cooling fluid used in cruise, then cutting power cables, and ending with the firing of the pyros that separate the cruise stage. It is the timepoint actions, positioned in time along the timeline, that make these configuration changes. However, GNC needs to know when such changes occur so it can apply the correct estimation and control parameters and algorithms. GNC inquires of the timeline when the actions it cares about have occurred; other actions, such as parachute deploy, are initiated by GNC which needs to inform the timeline when they are to occur. It is actually the timeline that deploys the parachute, but until GNC tells the timeline when that is to occur, the timepoint execution time is “undefined”. When GNC decides it is time to deploy the parachute, it tells the timeline “now” and the timeline then sets the parachute deploy timepoint’s *firing time* to the current time on the clock. Then the timeline executes the action just as it does for all other timepoints. After the parachute deploy timepoint’s firing time has been set, other timepoints’ schedule time becomes defined and their actions occur time-relative to parachute deploy. Figure 2 shows a timeline with timepoints and how the timepoints are ordered on the timeline.

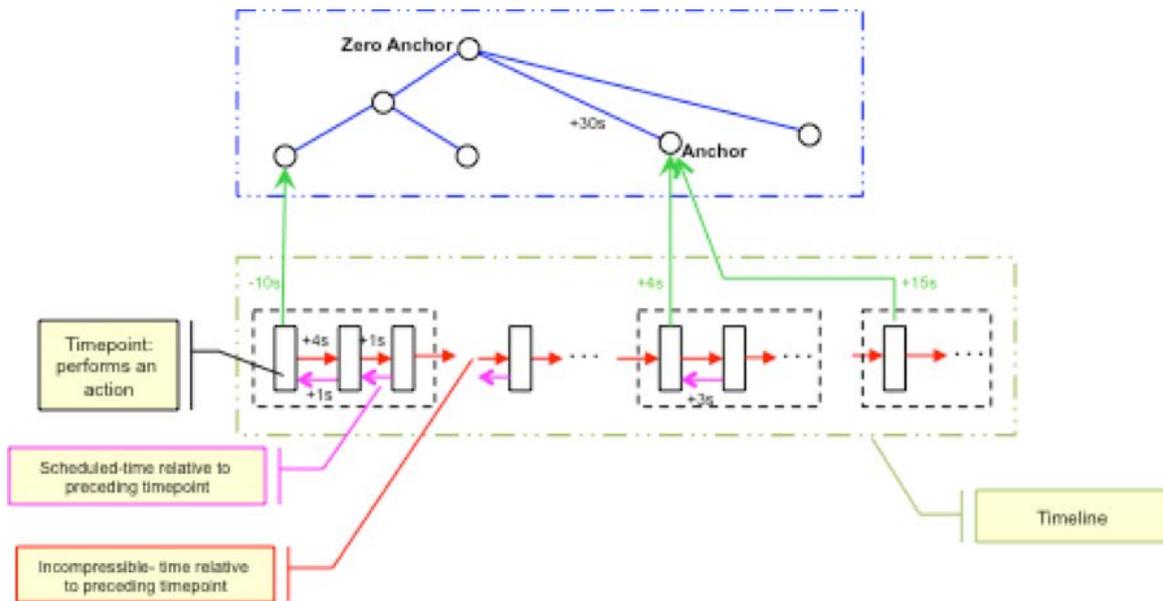


Figure 2. Timepoints are on a linear timeline; anchors specify certain external events.

Anchors, distinct from timepoints, are distinguished times, such as the navigated Mars entry point. Anchors are arranged in a tree so each anchor is relative to some other anchor, except for the root anchor whose time is defined to be zero. Any anchor whose time is an absolute time t (such as the navigated entry point time) is represented as an offset of t seconds from the zero root anchor. The anchor tree is represented in the program as an array of anchors topologically ordered so if anchor A is defined relative to anchor B, then anchor A follows anchor B in the array. In this way, a single sweep through the array allows all anchor times to be computed, since if A is at an offset from B, B's current value will already have been computed in the same timeline cycle.

In the software, a timepoint is a C structure, and the timeline is simply an array of timepoints. Each timepoint has a number of temporal constraints that must be satisfied in order to fire as shown in Figure 2: a minimum offset time from the previous timepoint, and either a desired offset time from the previous timepoint or from an anchor. Thus, a timepoint is relative to either an anchor or the previous timepoint on the timeline. The Edl timeline runs at 64 hz, and during each cycle the timeline interpreter considers whether to fire the "current timepoint". It looks at the current timepoint and computes a Boolean expression that asks if the current time on the clock satisfies either the desired offset time from the firing of the previous timepoint or the desired offset from the specified anchor, and is at least the minimum offset from the previous timepoint. If so, the timepoint fires by executing the function named in the timepoint structure. After executing the function, the timepoint changes state and waits for a reply that verifies that the timepoint action completed. When that reply arrives, or the reply timeout expires, the timepoint sets its "firing time" slot to the current time on the clock. The timeline then proceeds to the next timepoint.

With some limitations, relative-time offsets in the timeline or the anchor tree can be a positive or negative number, or can be "undefined". The "undefined" value encountered while evaluating an arithmetic expression results in "undefined". The Boolean logic in the Edl timeline code was generally simple, so interpreting "undefined" as False was adequate, such as in the above described timing offset requirements for timepoint firing.

In each timeline cycle, the anchor tree is computed first, and then the current timepoint is checked to see if it can fire. If so, the timepoint firing takes place (the associated action is executed) and if the timepoint is to wait for a reply indicating completion, the timeline is done for this cycle. If the timepoint action did not request a reply, the timeline goes on to check the next timepoint for firing. Ground commands may change the offsets at any time between timeline cycles. Any change to an offset propagates fully in the next timeline cycle.

Time is an extremely difficult "object" to get right. It needs to be measured, converted to different forms, transmitted from one hardware unit to another, and synchronized across hardware units, both within the spacecraft and the test beds. There is one interesting method we used in the flight software to make calculations with time a little less troublesome than might otherwise have been the case. The programmer uses a function `clk_1_hz()` to get the current clock time with a resolution of 1 second. In addition, there are other functions `clk_x_hz()` to get the same clock at different resolutions. The best resolution, for software, was 64 hz. These resolutions are all powers of two. We also determined that the range of a 64-bit floating-point number was large enough to represent all times that the spacecraft would encounter over its lifetime, allowing sufficient bits to represent 1/64 second as well. All software timing, and arithmetic operations on time, could then be done exactly using double-precision floating-pointing numbers. This eliminates any concern about round-off errors accumulating. In fact, this was the reason we chose for the hardware clock to tick at 64 hz. We were not so fortunate with sensor timings. For example, the imu is 200

hz which is inconvenient for any binary system. Nevertheless, the scheme avoided a lot of trouble for the vast majority of time calculations on the spacecraft.

The EdlBg Module

This thread runs when the radar is on. It batches the radar data into rolling averages over 5 seconds of data and sends the result to the navigation filter at 1 hz. This thread exists because at the time of design of the system it was unclear how long this thread might occupy the processor. It was simply not good planning to try and fit it into the 64 hz GNC loop. As it turned out, all the work probably could have been done at a 1 hz rate and run inline with the 64 hz navigation filter code – this would have removed one thread from the system. However, that approach was deemed risky and not worth taking for the sake of one less thread, as the disruption of creating the thread later (for example, if we discovered we could not make it fit into the 64 hz thread) could be significant.

The EdlComm Module

The Entry, Descent, and Landing Communications module (EdlComm) is responsible for orchestrating the collection, management, and telemetry of Edl data, both recorded and real-time. All Edl telemetry goes through EdlComm as shown in Figure 3. It is a principle of the design that all data is written to non-volatile memory during Edl, and the telemetry that goes to the ground is simply a subset of that which is collected. Making real-time telemetry a subset of the recorded data, rather than a separate stream, eliminates the post-landing correlation problem of two separate streams.

There are several data streams with specific purposes. Many important events go through the MFSK path (Multiple Frequency-Shift Keying) and are communicated in real-time using very low bandwidth X-band tones. MFSK is present in case of catastrophe or when normal communication is impossible (such as atmospheric entry). It allows one tone from a set of 256 possible tones to be sent at one time. A tone needs to be transmitted for 10 seconds to ensure correct decoding on the ground. The MFSK module selects the most important tone from a set of outstanding tone requests and sends it. When a tone request gets too old or has been overtaken by events, it is dropped from the candidate set. In addition, higher bandwidth channelized telemetry, event reporting, and data products are sent in real-time using a UHF link to an orbiting relay satellite. The frequency and set of real-time data products changes as the spacecraft progresses through Edl, and EdlComm is responsible for ensuring that the correct data is present in the UHF data stream based on vehicle state.

OVERALL EDL OPERATION

Edl begins operation several days before landing, in the phase called pre-Edl. System Fault Protect can still take action during pre-Edl, so to prevent Edl from interfering with these actions, System Fault Protect can inhibit Edl, that is, Edl will make no progress while system fault protect repairs a situation. After pre-Edl, which ends several hours before touchdown, System Fault Protect is no longer allowed to inhibit Edl.

The Edl timeline will not begin operation, and thus EdlGnc will not run, until the spacecraft receives two particular commands. The first command is “set edl parameters” which is a ground command with the latest navigation information. This command is sent each time there is a useful navigation update. After the Edl parameter command has been received at least once, the “do edl” command from the ground gives the Edl module permission to begin running the timeline.

The “do edl” command cannot be sent to the spacecraft until the spacecraft is in late cruise. These conditions serve to safeguard against accidental Edl execution.

The 64 hz clock interrupt handler sends a tick message at 64 hz to begin each 64 hz processing cycle. This tick message from the clock goes to the sync module that actually drives the Edl system. Sync receives the tick and, sequentially, sends a message to each of a number of modules: radar, imu, Edl, pyro, descent engines, and thrusters, in this order. These modules form a hard real-time rate-group such that, as each module finishes its work for the current 64 hz cycle, it sends a confirmation message back to sync, which sync then uses to move on to the next module in the list. This is how the modules are run in sequence. It would have been possible to arrange a more data-flow driven system, but for simplicity in testing it was deemed better to keep modules always running in the same sequence, regardless of their internal state. If a module is idle and it receives a 64 hz tick, it simply confirms and goes back to sleep. Sync is also where any cycle overruns are caught.

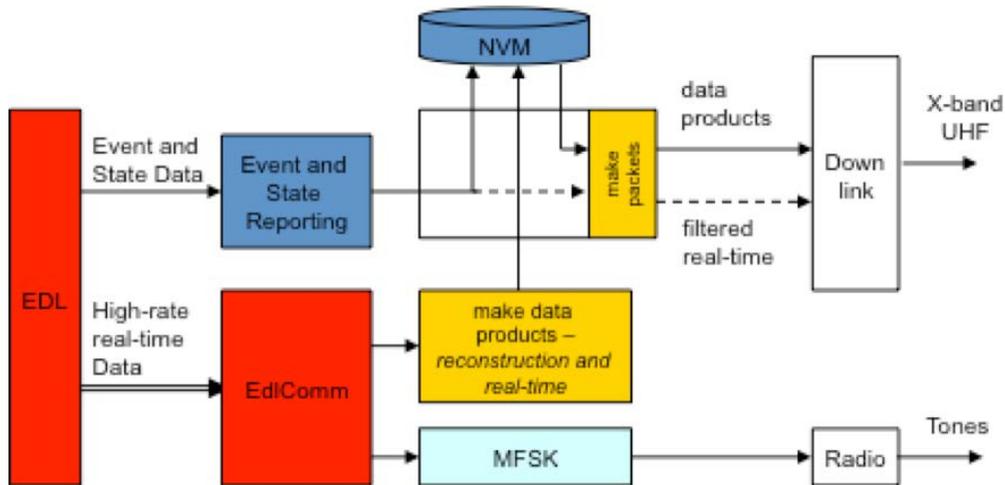


Figure 3. All data is recorded; some is filtered and goes to ground as real-time packets and/or tones.

Zero or more timepoints may fire in any one 64 hz cycle. Timepoints can be separated by zero time, so there is a limit of five timepoint firings per cycle so as not to overload the processor. After the timeline runs, Edl runs one GNC cycle so the estimators and controllers can do their work. Note that the timeline ran first, since it may perform an action that affects GNC during that same cycle. However, GNC may perform an action that affects the timeline. For this reason, the timeline runs again, immediately after GNC, during the same cycle. The timeline can actually run any number of times during one 64 hz cycle, but GNC can run only once.

There are some restrictions on the timeline and its timepoints:

- The timeline is sequential; there are no concurrent, parallel branches.
- All constraints are nearest neighbor timepoint, or to an anchor.

The constraints are reminiscent of a temporal constraint network, but the nearest-neighbor restriction makes the entire timeline sequential. More complex relationships could have been represented, but were specifically excluded, in order to simplify testing. The cost is that many relationships among timepoints are not directly represented. Instead, offline analysis ensured that if the explicit constraints in the timeline were met, then so were the implicit constraints.

The Edl timeline benefited a great deal from a Timeline autocode tool. System engineers developed the timeline using a spreadsheet with extensive analysis tools. After determining the best timepoint firing times and timepoint separation times, given the current state of knowledge of spacecraft and software behavior, the spreadsheet passed through the Timeline autocoder. The autocoder output was a large data structure that represented all the timepoints and their constraints. During execution, the timeline engine interpreted this data structure. The Timeline tool saved considerable time, as the timeline underwent well over one hundred versions even after it was “finished”. Test results were responsible for most of the adjustments.

The MER Edl allowed for parallel branches of timelines, and these were used in MER in some of the mechanical systems after landing, such as airbag pulling and solar array deployments where more than one mechanism was active at one time. Future systems may require more constraints and non-linear constraint networks. This might be necessary when alternative sequences of actions are needed, and more concurrent actions required, to complete work on time. This was simply not needed for MSL. A computer reboot during pre-Edl (before cruise-stage separation) resulted in the prime computer re-starting and Edl backtracking specified timepoints and repeating those timepoint actions. A computer reset after pre-Edl requires a different response since recovery is time critical. In this case, the backup computer runs a reduced version of the prime Edl software. This pared down version of Edl is called Second Chance, and was limited to the absolute minimum actions necessary to land safely.

SECOND CHANCE

The Edl software in Second Chance is the same as in the prime system, but where non-essential timepoint actions were replaced with no-operation. Also, some non-essential modules were completely removed from the Second Chance load, while a number of other modules were lightly or somewhat more extensively modified to work with Second Chance. Among the modules excluded from the Second Chance build are cruise attitude control and a large number of modules for surface operations, and all the science instrument modules. The rationale for this arrangement is that whatever caused the failure might be avoided by the backup version since fewer devices and program events are taking place. To support Second Chance, the prime Edl sends state data to the backup computer as it runs. Second Chance then picks up where the prime Edl left off. To ensure that the ground is aware of any transition to Second Chance, the tones emitted by Second Chance are different from those of the prime Edl system.

Perhaps the overriding design goal of Second Chance is that it “Do No Harm”. Extensive interlocks and testing ensured that Second Chance would not accidentally begin to execute the timeline or GN&C. This included a ground command required to enable Second Chance. Nevertheless, Second Chance was enabled during MSL Edl, but the signal for it to run never arrived. When touchdown occurred, it dutifully turned itself off.

TESTING THE SOFTWARE

Figure 4 shows the test flow for Edl flight software. In all test venues, the same Edl flight software is used. The only difference in each venue is the fidelity of the test and which non-Edl software modules and hardware units are present, and which are simulated.

The GNC team wrote the EdlGnc and EdlBg modules, and tested them in the GSTS test bed, a Linux system with no hardware in the loop. The intention here is two-fold: simple regression tests to ensure the latest GNC software changes did not cause any previously tested scenarios functions to fail, and as new models became available, rather sophisticated Edl GNC tests were run. This test bed, however, played no role in software performance or margin testing, as com-

puter execution time was not modeled. An additional test bed was at NASA Langley where Monte Carlo POST runs were made for the very detailed performance analysis of the Edl flight software, utilizing many models of atmosphere, terrain, and so forth. This test bed was nevertheless a unit test bed since, as much as possible, only the EDL flight software was present.

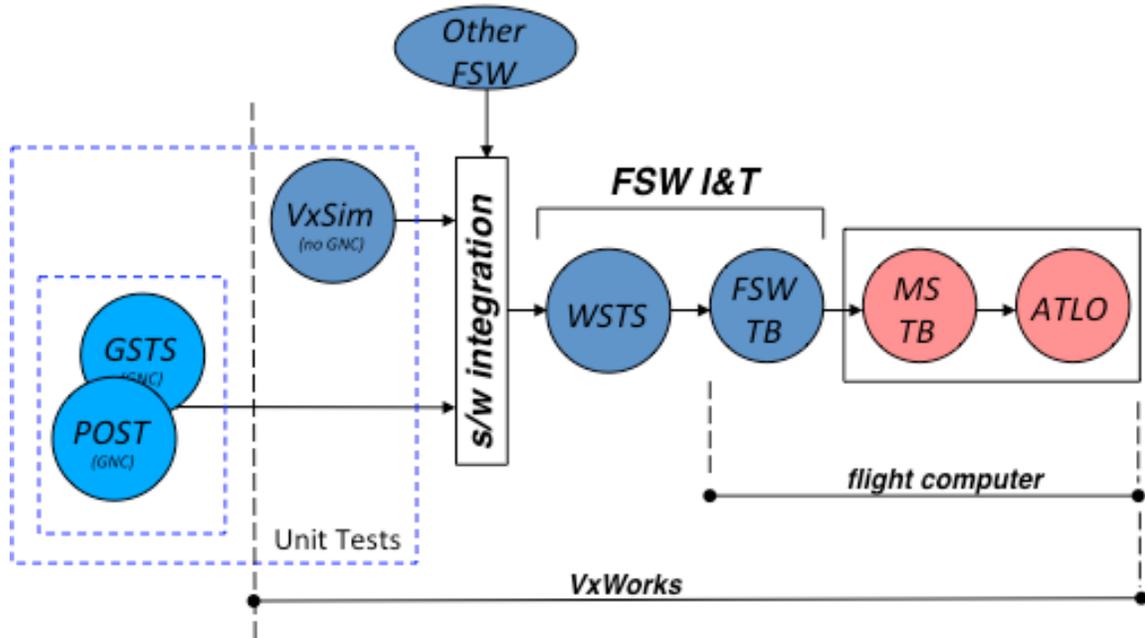


Figure 4. The testing venues for MSL Edl Flight Software

In parallel, the non-GNC portions of the Edl flight software ran in a separate environment analogous to GSTS for GNC. Here too, no hardware was present and computer execution time was not modeled. These two environments were the unit test environments and were very effective at removing any errors in the modules themselves.

The WSTS environment is a Linux system where all the flight software can run together, but still without computer performance measurement. The FSWTB is the first place where computer performance can be measured. The FSWTB had a RAD750 and significant GNC hardware-in-the-loop. For example, a gyro, thrusters, and so forth, but where the signals were generated by support hardware but injected into test ports in the hardware devices to simulate as close as possible real spacecraft behavior. However, additional software debugging tools and internet connections made the FSWTB easier to use.

In addition to design, code, and test reviews, every flight software module went through at least one code quality review. Several commercially available static analysis tools were used to improve code quality, and a JPL tool called Scrub coordinated the running of the analysis tools, collected the results, and assisted in the recordkeeping during code quality reviews. Also, Valgrind and purify were required test tools for all flight software modules. In addition, Edl underwent additional reviews by non-Edl developers from the flight software team, and to review by a team at the NASA IV&V Center in West Virginia.

Any software developer could use any of the test environments mentioned so far. The MSTB was an even higher fidelity test bed that had more hardware than FSWTB, but ATLO was the end test done on the actual spacecraft.

LESSONS LEARNED

- Complex code needs an “Explanation System”: This means that it can be difficult for a tester or flight controller to understand the behavior of the system when complex conditions occur. It can even be difficult for the designer and programmer to remember the intricacies of a system as large as MSL and more than once we wasted time looking for incorrect behavior that did not happen. When a program goes beyond a few conditionals, and it is a “rare” situation, there should be messages to the ground that explain the conditions that resulted in the unusual, but correct, actions taken by the software.
- It can be useful to have redundant code that checks conditions, even if it does nothing about them: For example, the time constraints in the Edl timeline were “rollups” of sets of other constraints bookkept by the system engineer. Those constraints should be included in the code, and checked, but necessarily raise an error as much as simply report that some sub-constraint was not satisfied, even though the overall “roll-up” constraint was satisfactory. This helps finding vulnerabilities, and might be considered a special case of the previous point.
- Autocode should parse source code, as well as simply scan: MSL made extensive use of autocode – specification files that were translated into C code. Sometimes C source was scanned for inputs, but scanning is not sufficient. Simpler specifications, with less programmer effort and better generated code covering more circumstances would have been possible if C source files were parsed whenever information needed by the autocode tool could be gathered from the source. A good deal of the complexity of the autocode system used on MSL was due to not making the autocoder a real parser.
- Designs tradeoffs are difficult, and if made on the basis of cost, make sure you have all the costs identified: For example, it is easy to say “we should do it the way we did it before: we have the designs and the same people”. When margin gets thin, and you need to compress and make optimizations, it can cost far more than one can imagine without a thorough analysis.

ACKNOWLEDGEMENTS

The author has worked with Miguel San Martin, Paul Brugarolas, and Fred Serricchio of the GN&C team on several projects. Along with Martin Greco, EDL system engineer, these are among the most outstanding individuals with whom the author has ever worked. Ben Cichy led the Flight Software Team on MSL for most of the software development period. I thought MSL was large enough that no one person could understand every element of the entire system, but that may not hold for Cichy. I also wish to acknowledge the contributions of Joe Snyder, the first lead, who knew just how big a job it was going to be.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

¹ A.M. San Martin, S.W. Lee, E.C. Wong, "The Development of the MSL Guidance, Navigation, and Control System for Entry, Descent, and Landing," AAS/AIAA Space Flight Mechanics Meeting, Kauai, HI, Feb. 2013.

² F. Serricchio, A.M. San Martin, E.C. Wong, "Mars Science Laboratory Navigation Filter," AAS/AIAA Space Flight Mechanics Meeting, Kauai, HI, Feb. 2013.