Flight-Ground Integration: the Future of Operability

Dr. Christopher A. Grasso[1]
Blue Sun Enterprises, Boulder, Colorado, 80302

and

Patricia d. Lock[2]
Jet Propulsion Laboratory / California Institute of Technology, Pasadena, California, 91109

Flight and ground integration is an ongoing challenge in the development of deep space mission operations systems. Separate development teams and schedules exacerbate the problem. Enhancing the operability of the flight and ground interactions has proven to be a strategy that reduces both cost and risk. One of the first areas of operability addressed has been the commanding and sequencing system. Virtual Machine Language (VML) sequencing was developed to improve operability and has now been used on fifteen NASA deep-space missions, most recently on NASA's Mars Atmosphere and Volatile EvolutioN (MAVEN) mission. Onboard adaptation to new missions has taken the form of data-driven configuration of the VML software for command and variable definitions without changes to underlying flight code, dramatically reducing cost and risk. Continuity in capability has allowed new VML mission sequencing to be based on products from prior missions, which are then modified as necessary to accommodate different hardware capabilities. In conjunction with the NASA SBIR "Reactive Rendezvous and Docking Sequencer", improvements incorporated into the various versions of VML since 1998 allow VML scripts to perform much of the work that formerly would have required expensive flight software development. The latest version, VML 3.0, has been enhanced to include reactive state machines for autonomous management of critical spacecraft operations, object-oriented element organization, and matrix/vector operations.

I. Mission Operations Domain

The mission operations domain of a deep space mission encompasses both the development of the system used to carry out operations, and the conduct of the operations phase of the mission itself. A Mission Operations System (MOS) is comprised of a Ground Data System (GDS), which includes the software, hardware, networks, facilities of the domain, plus the MOS teams, policies, processes, procedures, and training. Figure 1 shows the basic operations functions carried out for a deep space mission.

It is essential to view deep space mission operations as a function of the mission objectives. Seen in this way, operations can be thought of as consisting of three items, in priority order:

1. Collect science data (to achieve the mission objectives)
2. Operate the flight system (to collect science)
3. Build, test, and deliver the flight system to orbit (in order to have a platform to operate)

In order to achieve the highest priority, operations personnel must successfully operate the flight system to collect science data. The more operable the flight system, the simpler and more reliable the operations team's job becomes. During development, however, it is easy to lose sight of mission objectives in the drive to build and launch the flight system. What may be lost is a focus on the achievement of the mission objectives that can only be accomplished via operation of that system. Operable flight systems are more likely to reach the objectives, and keep cost and risk down as well.

[1] VML Principal, Blue Sun Enterprises, 1942 Broadway Suite 314, Boulder, CO, 80302, Senior MemberAIAA.
[2] Group Supervisor, Mission Operations System Engineering, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA, 91109, Member AIAA.

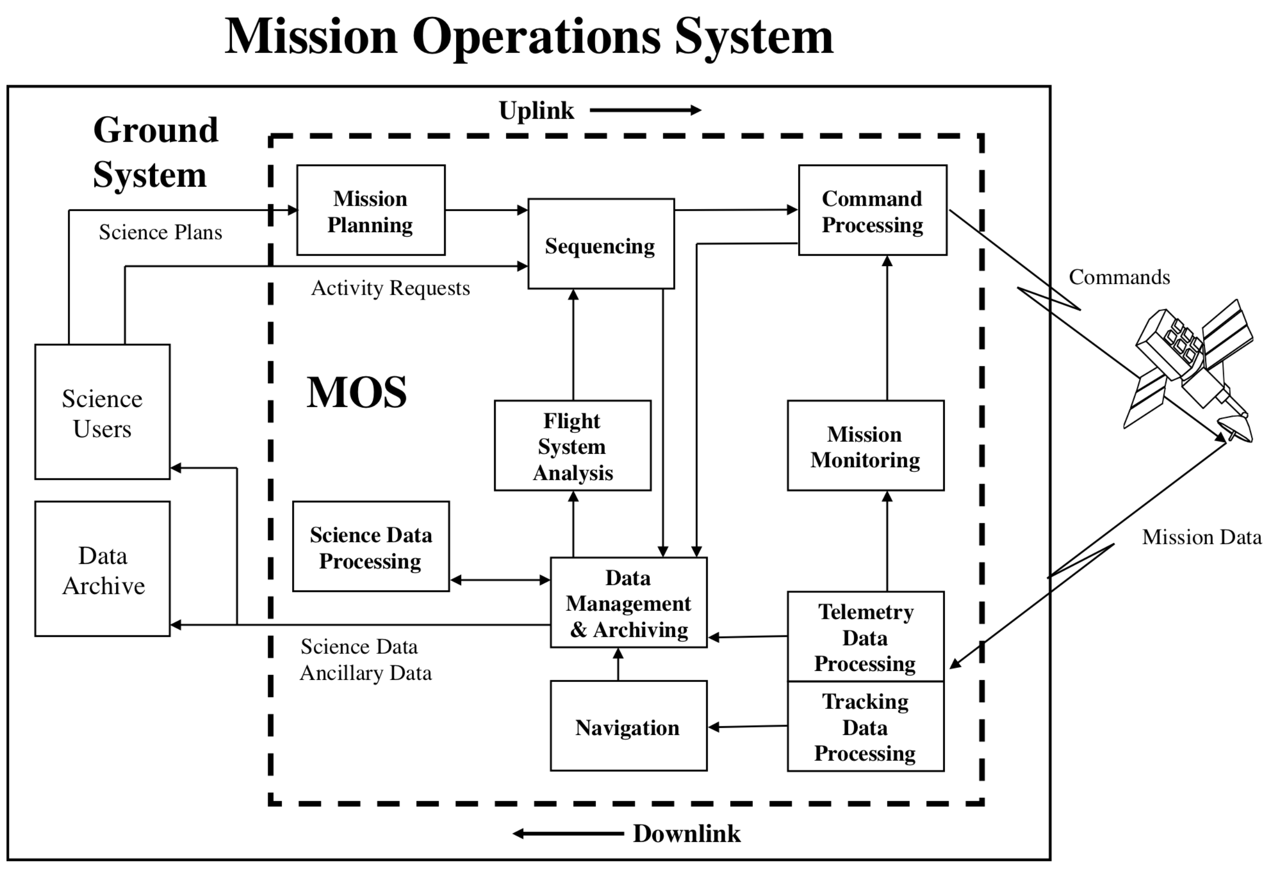

Figure 1. A simplified view of mission operations functions, showing the flow of data through the system

II. The Problem: Low Operability

A common practice during the development of a mission's architecture is to shape the Flight and Mission Operations Systems as if they are separate, stand-alone systems. This practice omits operability – purposeful design of the flight system to enable operations to be as uncomplicated and trouble-free as possible. Rooted in the long-held belief that ground and flight systems have little interaction beyond the uplink and downlink external interfaces, the flight system and ground system development processes are often largely decoupled. Though this may have been sufficient with simpler spacecraft, the advent of complex flight and ground software suites has made this approach inefficient, expensive, and high risk.

A. Operability's effects

In the design of modern flight systems, operability considerations are uneven. Conversations with Mars Reconnaissance Orbiter (MRO) development engineers have brought to light examples of both strong and weak operability design in the mission.

One design choice led to two instruments sharing a single interface to the command and data handling system. When one instrument of the two caused a data overflow problem, both instruments had to stop taking data while the anomaly was investigated.

Another example involves the MRO onboard data system. Its mass memory had been designed to be extremely large, compared to earlier Mars missions, in order to support the volume of data MRO was expected to generate. Normally, this would be an excellent operability choice, providing flexibility in implementing data return strategies. However, the flight computer processor was not fast enough to easily manage the large number of data items that the data volume contained. Onboard data store management created significant unplanned workload for the operations team.

In contrast, the MRO power system is an example of an excellent design that was focused on operability. Early in the design phase, MRO designers made power system operability a high priority and assigned trades to investigate its design space. As a result, the power system was built with sufficient margin to obviate the need for daily or weekly power management, relieving the operations team of that common task.

The deficiencies of a poorly integrated flight and ground system typically become apparent in the integration and test phase when uplink, downlink, and system end-to-end tests are conducted. A test-as-you-fly philosophy also serves to uncover flight-like problems during system integration. Examples of typical disconnects include sequence size overruns, memory/storage allocation problems, bus errors and command timing constraints, and out-of-tolerance flight system behaviors. Flight system idiosyncrasies also surface in this period, adding to changes during the run-up up to launch.

At this point in development, the flight system is limited to critical changes, driving the standard decision to move workarounds into operations. Once responsibility has been transferred to the operations team after launch, the compromised flight system design continues to introduce cost and risk. As shown in Figure 2, system deficiencies frequently require unanticipated (and unbudgeted) development cost, extra personnel time, and the need to undertake unnecessarily complex activities on a recurring basis.

B. Operations crossover

The solution to these problems is to consciously design flight and ground systems together. In that way, reasonable trades can be made to decide the most effective implementation of operability features, and to budget accordingly. In some cases, allocating certain operability challenges to the operations phase may be more effective, provided there is time to automate solutions. In other cases, operations considerations in early flight system design can lead to operable systems with no additional cost. It is a matter of considering all of the impacts in design trades, rather than only the traditional spacecraft-centric impacts. With complete trades, the results can be analyzed and implemented by the mission team as a whole.

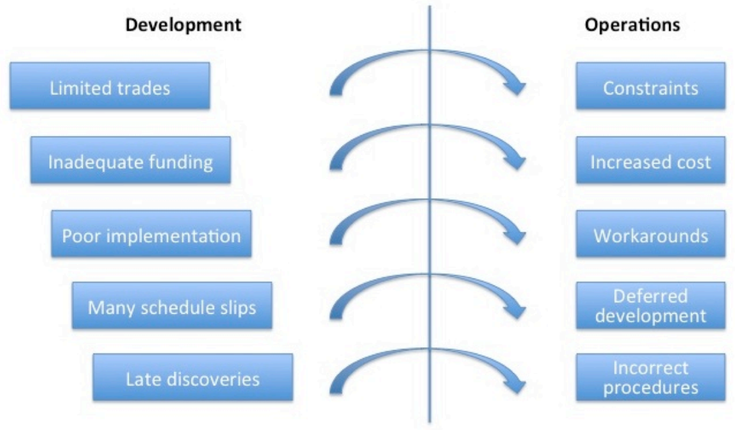

Figure 2. Effects development problems have on operations after development transitions to operations

One of the simplest ways to improve operability is to shift some functions, normally done repetitively on the ground, to the flight system. When done effectively, this can increase system flexibility and robustness. One such approach is via the Virtual Machine Language (VML)[1-9, 15, 16] sequencing system.

III. Virtual Machine Language for Operations

VML is a processing language tuned to the needs of spacecraft operations. Before VML, operators had little influence over the way their spacecraft responded to commands, and a very limited set of commanding and sequencing capabilities. Anomalous or idiosyncratic spacecraft behavior was dealt with via onerous and sometimes risky workarounds, or by costly flight software changes.

VML provides functionality to allow operators to implement onboard reusable solutions that in the past could only have been implemented in flight software. VML avoids many of the problems associated with typical flight software source code developed in C, C++, or Ada, while providing flexibility to implement straightforward operations. By placing operations onboard rather than using ground-based commanding scripts, VML constructs can also make decisions locally, reacting to environmental conditions seen by a spacecraft minutes or hours away from earth in round-trip light speed delay. By using VML for these onboard capabilities rather than flight software, late changes and idiosyncrasies can be accommodated at minimal cost and risk.

VML provides several methods for commanding and sequencing missions, including sequences, blocks, objects, and state machines. VML does not run on the "bare iron" of the host microprocessor. Instead, the language is implemented as a byte code binary, and is interpreted at runtime by onboard software known as the VML Flight Component. This approach provides a safe sandbox for execution, eliminating many common problems found in flight software implementations. VML provides operations structure for issuing spacecraft commands, while protecting the user against problems encountered with low level languages like C, including accidental assignment, off-the-end array access, division by zero, type coercion, and missing functions. Ultimately, VML constructs issue the spacecraft commands required to complete tasks using logic based on time and the conditions present. VML operates the flight system within the intended constraints and tested capabilities of the flight software, just as the ground system does. It provides a growing set of features based on mission requests and has been used on fifteen missions, with more in development.

A. Spacecraft commanding

Commands are directives to the spacecraft, typically represented in a human-readable form and translated to a binary format. Commands cause the spacecraft to behave in some desirable way for the purposes of science collection, power management, thermal stabilization, propulsive maneuvers, pyrotechnic firing, and so forth. Commands may originate from ground-based human operators, flight software elements, or sequences.

Sequencing is the issuance of spacecraft commands from an on-board store that allows the spacecraft to perform in an automated fashion when no uplink is available, or when light speed delays preclude direct commanding from the ground. Commands in VML may be timed according to absolute (wall-clock) time and relative time, as well as in response to conditions on board the spacecraft using a technique known as *event-driven* sequencing.

B. Heritage

Virtual Machine Language development began in 1997. Five versions have been implemented. VML has been used or is in use on 15 NASA flight missions to date, including Stardust[2], Genesis, Mars Odyssey, Spitzer Space Telescope[3, 4, 8], MRO, Dawn, Phoenix, Juno, GRAIL, MAVEN, OSIRIS-REx, InSight, and the Resource Prospector Mission RESOLVE instrument package. A timeline of VML use appears in Figure 3.

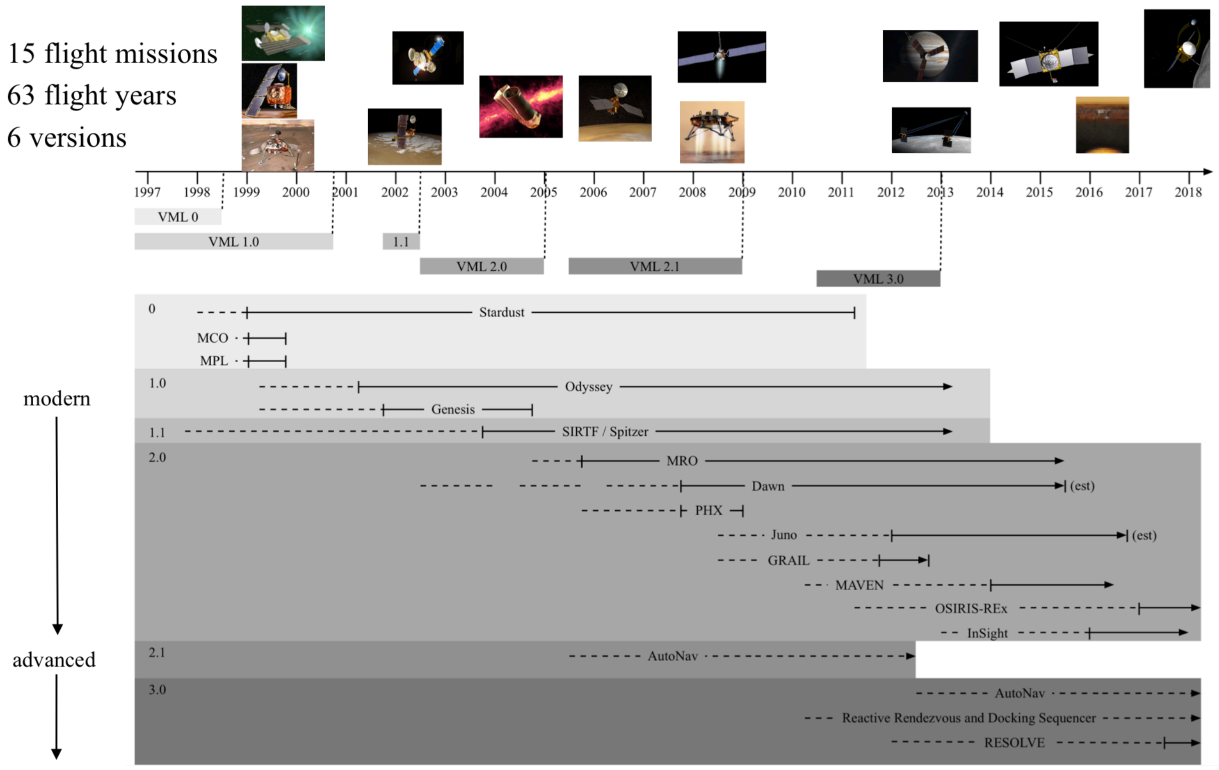

Figure 3. VML heritage timeline, 1997 - 2018, showing NASA deep-space missions and their VML versions

C. Feature list evolution

The VML flight environment provides multiple threads of execution within one operating system task using a data-driven construct known as a *sequencing engine*. VML allows an extensive set of variable types, including integers, floats, Boolean values, strings, and arrays. Arithmetic and trigonometric calculations, logical manipulations, and matrix operations are available for use. Conditionals may be used to make decisions based on local values at runtime. WHILE and FOR loops perform iteration. Sequences exist as named functions, which can accept parameters and have local variables. Functions may be packaged together into a single file loaded onto an engine in order to associate runtime behavior or to provide libraries of commonly needed services. Objects with methods package code and data together. Specialized objects called *state machines* provide a directly executable set

Table 1: VML Feature List Evolution

	pre-VML	VML 0	VML 1.0	VML 1.1	VML 2.0	VML 3.0
Processor	Custom	PPC, Sparc	PPC, Sparc	PPC, Sparc	PPC, Sparc	PPC, Sparc, Intel
Sequences	X	X	X	X	X	X
Integer data	X	X	X	X	X	X
Reusable blocks	X	X	X	X	X	X
Arithmetic, bit, comparison operators		X	X	X	X	X
Block libraries			X	X	X	X
Floats, unsigned integers, booleans, strings			X	X	X	X
If / Else If / Else conditional			X	X	X	X
While loop			X	X	X	X
Wait on single global variable			X	X	X	X
Onboard command generation			X	X	X	X
String table (uplink size reduction)				X	X	X
For loop					X	X
Engine sizing					X	X
Trigonometric functions					X	X
Optional time tags						X
Command completion						X
Object-oriented command / function syntax						X
Human-readable ground commanding						X
Matrix and vector operations						X
Heterogeneous arrays						X
Wait on complex conditions						X
Select loop for triggering rules						X
State machines						X
Synchronized state transitions						X
Unix-hosted definition tools, web database						X

of reactive actions, and can intrinsically coordinate to perform sophisticated autonomy as an expert system. Table 1 shows the evolution of VML features over time.

D. Simple tool chain

The VML tool suite consists of an embedded VML Flight Component (VMLFC), a ground-based VML Compiler, the Offline Virtual Machine (OLVM) program, and the VML Command Translator. This suite allows products to be generated, loaded, executed, and tested. The relationships among these VML tools are shown in Figure 4. A source file containing human-readable VML script is generated using a standard text editor or other tool. The compiler translates the text file into a loadable binary file, translating commands and times using mission-specific tools and tracking valid global variables and symbolic constants for the mission. The file produced can then be loaded by the VMLFC.

Typical development processes run the module under OLVM in order to test and validate the behavior of the code. User-defined tests can be generated with very little effort by capturing a user-guided session and then rerunning the test using the extracted user keystrokes from the session. OLVM can be widely deployed on Linux, Macintosh, and Sun platforms, allowing products to be fully tested before validating them on flight-like hardware in the less-accessible real-time Software Test Laboratory.

Mission adaptation of the onboard flight component requires the definition of spacecraft commands and sequencing global variables. This task is performed by systems engineering using the VML Database, then translating those database elements to various definition files compiled into the flight software and elsewhere using the VML Configuration Generator. Products created by this process are then incorporated into the VML Flight

Components in the same manner across all platforms, including OLVM, the Software Testing lab flight software load, and the spacecraft flight software load.

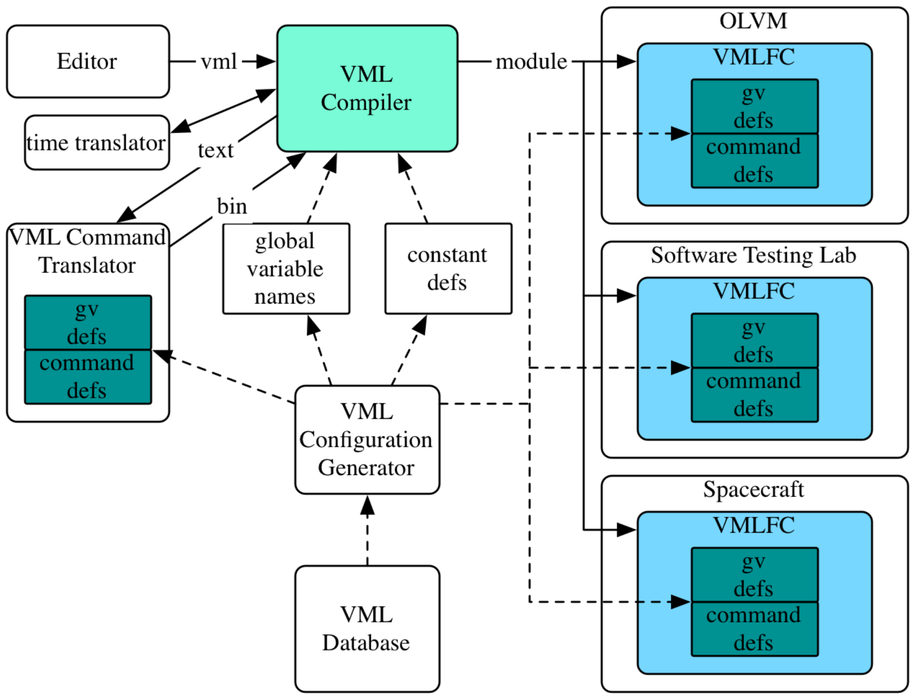

Figure 4: VML tool chain. Human-readable VML translated by compiler into format usable by workstation OLVM and a flight computer, either in a test lab or on the spacecraft.

E. Reusable blocks and master sequences

One of the most commonly used features in VML is the reusable *block*. Blocks are named functions that allow input parameters, and permit only relative timing of statements. This relative timing allows blocks to be reused without changing time tags within the element. Missions can then write blocks for a wide variety of repeated activities on the spacecraft, including reaction wheel desaturation, uplink and downlink initiation and termination, instrument control, aerobraking[8], trajectory burns, pointing, battery management, etc.

Once a block has been developed and tested, it can be used as an abstract capability. Blocks are built and tested according to requirements and flight rules via a user-friendly process similar to (but much simpler than) a flight software development process. Once tested, there is no need to scrutinize and review a block's contents on every invocation. In a sense, blocks act like super-commands for the spacecraft, providing more functionality and flexibility than simple spacecraft commands.

Blocks are frequently grouped together and stored on an engine as a *library*. Engineering and science libraries deal with spacecraft housekeeping and instrument control, respectively. Dividing blocks into two separate libraries allows the spacecraft operations and science teams easier control over their respective products. Since the libraries are always present in an engine, the blocks are immediately available to be started via ground command, or to be called from other blocks and sequences.

A *master sequence* is a single-use sequence containing absolute time tags that is built to execute during a known time period. The sequence team typically collects requests from the engineering and science teams to schedule specific activities aboard the spacecraft, and generates a master sequence to implement an integrated schedule. Master sequences typically span anywhere from twelve hours (as on Spitzer) to two weeks (as on Mars missions featuring VML).

Most of the activities within a master sequence involve invoking blocks rather than issuing spacecraft commands. This simplifies generation of the master sequence, dramatically reduces the time needed to review the activities, minimizes the size of the file that contains the master, and reduces the probability of violating flight rules or other requirements.

Master sequences may be chained together, wherein the last statement of the current master loads the next known master by file name. By following a standardized naming convention, the current master sequence is able to start its successor so long as the successor is loaded onboard sometime before the end of the current master sequence. In the case of short-duration master sequences, multiple masters may be stored onboard each day. For longer duration master sequences, new masters may be placed onboard as infrequently as once a month. The approach taken is spelled out in the documents defining the Mission Operations System.

F. Objects and state machines

VML 3.0 employs new ways of organizing logic and data over its predecessor versions. First among these is the *object*. An object combines data and functions into a cohesive package. In doing so, data are represented as attributes within the object, which is persistent and shared among the functions (called *methods*) of the object, but hidden from direct manipulation by other objects or blocks. An object acts like a subset of a block library, but contains functionality relevant to a more specific domain, rather than having all functionality needed by the entire spacecraft. As such, an object may be developed by instrument or subsystem personnel, and may be easily updated without affecting other objects in the system.

Objects allow operations developers to abstract capabilities into VML code intended to deal with just one subsystem: uplink, downlink, thermal control, attitude control, etc., become candidate objects to be coded. When operations for a new mission are undertaken, a previous mission's objects may be cloned then modified to reflect the particulars of the new mission. Objects provide a useful abstraction, allowing detailed changes to be isolated within individual methods, while maintaining method names that match the preceding mission. In this way, a pre-built structure for operations is applied from one VML mission to the next. Objects enhance operability by providing a convenient abstraction mechanism that can be applied from mission to mission.

A specialized object called a *state machine* has been added to VML 3.0. A state machine is a highly organized way of constraining activities within the sequencing domain to behave as a series of named states transitioning to other states based on conditions. This causes the system to behave according to tightly defined specifications, and avoids accidental violation of requirements. Using coordinated state machines allows complex problems to be broken down into simple, testable elements that feature simple operational transparency, and allow the system to be easily changed and extended.

VML state machines have a graphical representation very similar to that of Unified Modeling Language (UML) used in software engineering, but include the ability to synchronize transition-taking behavior among separate state machines. This synchronization ability allows a set of state machines to coordinate their actions by design, and act together as an expert system to accomplish a goal.

VML state machines have been used to demonstrate comet and asteroid touch-and-go missions, lunar landing missions, Mars sample return automated rendezvous and docking, coordination of instrument activities on the RESOLVE instrument package for lunar regolith characterization and oxygen production, and a variety of onboard autonomy applications. An example state machine for performing a hypothetical Mars Sample Return mission appears in Figure 5.

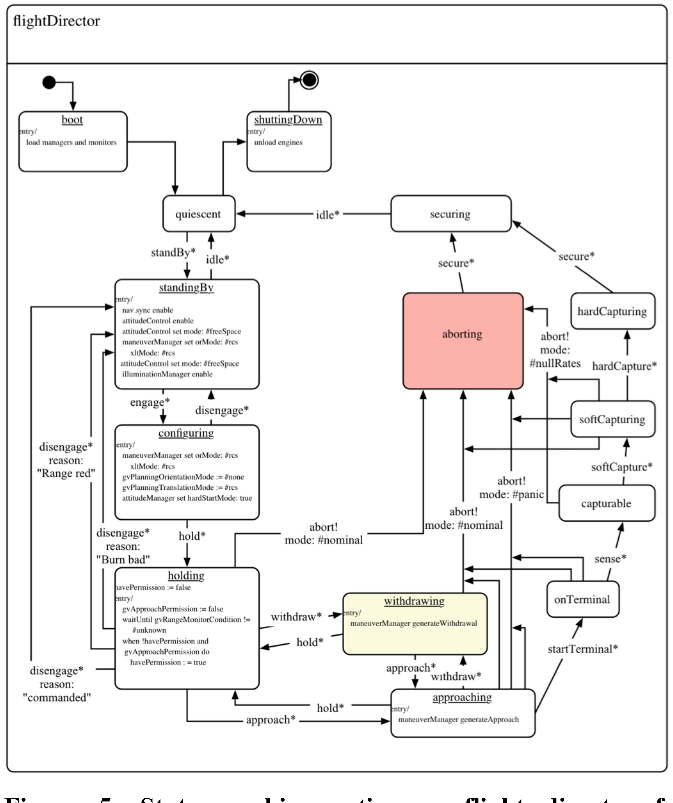

Figure 5: State machine acting as flight director for controlling a hypothetical Mars Sample Return mission

IV. How VML Addresses Spacecraft Operability

In order for VML to enhance operability, it was designed to be complete, yet adaptable. VML provides a common feature set on which to overlay needed operations functions. One of the hallmarks of an operable system is flexibility. VML allows a wide range of flexibility in the commanding and sequencing of a mission. Sequence constructs such as blocks allow highly repetitive operations (e.g. instrument power on), moderately flexible operations with some changeable parameters (e.g. a DSN contact), and highly customized elaborate operations using complex logic and event-driven execution (e.g. EDL activities.) This level of flexibility allows missions to choose and constrain the level of complexity that best suits their needs for each purpose.

VML works via three patterns: the flight component/flight software pattern, the ground/operations pattern, and the overall sequence design pattern. As a mission is defined, its VML needs are refined and adaptation begun. When the VML design patterns are used, much of what a mission needs already exists in the basic adaptation and the test suite delivered with the software. This results in lower effort, cost, and risk compared to non-standardized systems.

A. Flight software pattern

The flight component of VML is readily adaptable, based on mission needs. A number of different classes of missions have flown with VML, so a number of stock adaptation choices are available. Commands for VML are standardized across missions and a basic set of parameters, e.g., number of engines, size of instruction space, number of global variables, and other parameters are selected for the early development period. As the design of the overall mission progresses, changes to the parameters can be made based on better understanding of the mission's needs.

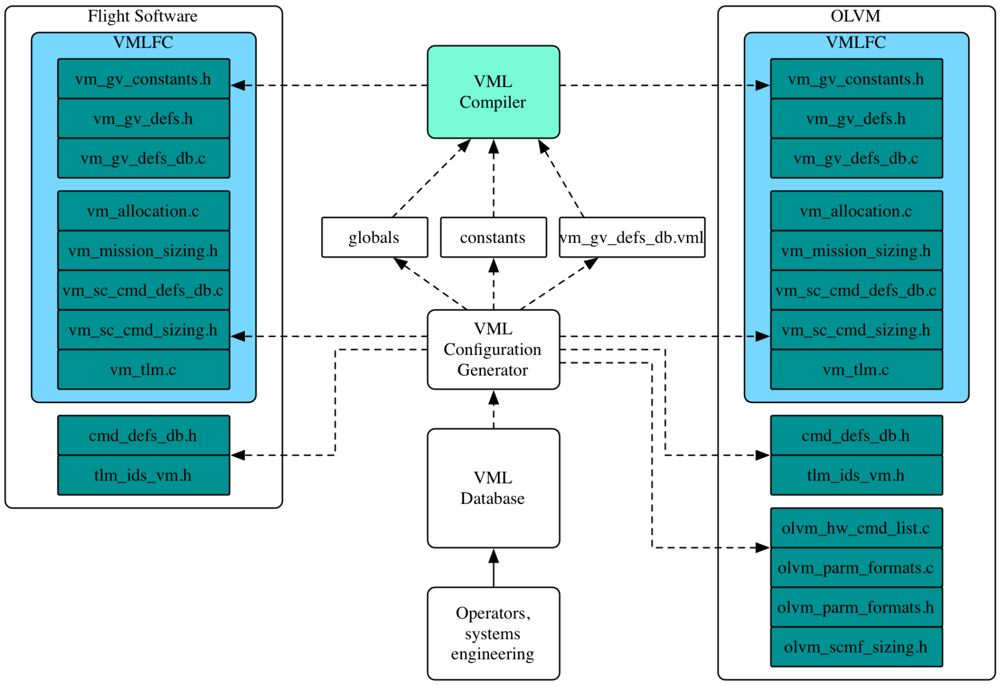

Figure 6: Tools to define mission-specific adaptations without hand-editing flight software elements

Depending on the mission, other changes may be required, either to the data-driven adaptation or to the core code. Changes made in the past have included adding the ability for VML to build commands at time of execution [Spitzer, Odyssey], interfacing with a spacecraft that has no file system [Dawn], addition/removal of CCSDS headers [Dawn], and inclusion of boilerplate instrument command formats [MRO]. VML has also been used on top of legacy systems to upgrade the capabilities of the sequencing system without disturbing the underlying heritage sequencing flight software [Dawn]. More recently, VML 3.0 has been augmented to allow dispatch over a network messaging system called Data Distribution System [Resource Prospector]. Once incorporated, the new capabilities are available to subsequent missions.

The architecture of the VML Flight Component allows mission-specific changes to be readily incorporated due to the layered nature of the code implementation and the segregation of mission-specific code and data from mission-generic code. Support tools provide mission-specific adaptation without the need to hand-edit files. These tools include a database for specifying commands, sequence global variables, internal sizing, number of engines, etc. Database outputs are translated by a separate application into .c and .h files compiled into the VML binary objects included in the flight software build. Operators and system engineering teams define the entries in the VML Database, and then run a production process that invokes the VML Configuration Generator. Global variable definitions require the additional invocation of the VML Compiler. The various .c and .h files compiled into the flight software load and the OLVM load are show in Figure 6.

B. Operations pattern

The operations pattern of VML begins with adaptation. After the overall flight software pattern is defined, operations "styles" can be selected and mission-specific adaptation begun. These styles vary widely among missions, based on their most critical needs. Questions to be answered in this phase include:

- What types of operations activities are highly repetitive and how many of them are there? Which activities require or benefit from an automated response? These questions scope the types and number of reusable blocks.
- Must there always be a sequence running on the spacecraft, or are gaps allowed? If sequences must always be running, then they must be chained together, one sequence starting the next, and fault protection must be ready to respond in the case of a sequence abort.
- Is all of the commanding based on specific time tags, or will some commanding be event-relative? What are the types of event-relative activities? How are the triggering events identified? Commanding all activities relative to time tags is the most common choice. Event-relative sequencing is much more powerful, but the non-deterministic commanding adaptation must be carefully controlled and tested. Timing and events scope the complexity of the blocks and their test program, and the number of global variables needed.

These and other decisions adapt VML to the mission, while incorporating the experience of other missions that have used it.

C. Development pattern

The earlier a mission chooses to use VML, the better the integration. Proposals benefit from the choice by fully defining the flight-ground interface as early as Step 1. With a well-understood set of configuration and sizing parameters, significant portions of both the flight software and ground system proposal sections are readily captured. Ground system costing using VML is well understood, whether the proposed mission is simple or highly complex. Operations via VML are also well understood and easily scaled, which is not true of missions using unique low-heritage flight software. VML is less expensive to implement and test than low-heritage systems, making the proposal description and cost analysis stronger and more credible.

In Phase A, VML provides the basic infrastructure of the flight/ground interactions. The focus is on choosing which options and features best fit the operations concept. Time is saved to work less common design questions, since so many of the flight-ground interaction questions have been answered by the choice of VML.

In Phase B, instrument and science needs and constraints begin to be addressed, enabling an interactive operations design. VML configuration parameters are defined based on the complexity of the mission's sequencing needs. In addition, rules on the mission's use of VML are defined. This constrains both development and operations complexity by limiting use of VML capabilities to areas where flexibility is needed for operability and risk reduction. Each mission is recommended to perform and document this exercise in phase B to ensure that all users are aware of project constraints. Mission-imposed constraints set in Phase B, however, can be changed throughout development and even into operations.

In Phase C, block development begins. The VML flight component continues to adapt to any mission-unique needs and is synchronized with the main flight software. VML is implemented in the spacecraft, the ground system, and each of the testbeds. VML is also implemented in any flight or ground simulators used for instrument pre-integration testing.

As integration commences in Phase D, the sequencing system is used for commanding the partially and fully integrated flight system. All system level tests, including system, scenario, and performance testing employ VML for commanding. While continuing block and sequence development, the ground system also conducts operations verification and training exercises, most of which are commanded using VML. This provides true test-as-you-fly capability and implementation.

This development pattern, refined over many missions, is highly reliable and predictable, allowing stable staff and resource estimates, and validated schedules. This saves on implementation cost, lowers schedule and functional risk, and provides a stable base capability on which to build. This stability in turn simplifies the implementation of support tools such as planners and data management systems that work in concert with the sequencing system to achieve the mission objectives.

V. Example Flight-Ground Integration using VML

The earliest missions to employ VML had a steep learning curve but greatly increased flexibility in designing their operations. Examples of a few of their accomplishments are described below.

A. Spitzer Space Telescope

The Spitzer Space Telescope team developed many of the earliest VML operations concepts, but due to a number of technical difficulties, Mars Odyssey launched first. Spitzer absorbed the excellent early lessons from Mars Odyssey operations and built on them for their more highly constrained needs.

One of Spitzer's instruments had been developed to handle most of their "sequencing" internally, passing instruction sets through the spacecraft command system with no interaction. This led to long, unwieldy commands and an unmanageable uplink volume. Due to Spitzer's tight observation-to-downlink pattern (11.5 hr. to 0.5 hr.), there was insufficient uplink time to load the instrument's commands. Using VML blocks to build and issue the troublesome instrument commands onboard allowed the mission to reduce uplink volume by 90%.

Figure 7 illustrates a typical master/slave arrangement similar to the one used on Spitzer. Master sequences are generated by a planning tool with inputs from principal investigators and spacecraft operators and placed onboard on a short cadence. The master sequences invoke slave sequences and onboard blocks during an observation period, followed by activating downlink services at the end of the block, after which the next master sequence takes over. Slaves are repeatable relative-timed sequences that may either be stored onboard or may be uplinked from the ground. Blocks are repeatable code chunks stored in onboard libraries which serve as super-commands performing tasks requiring logic.

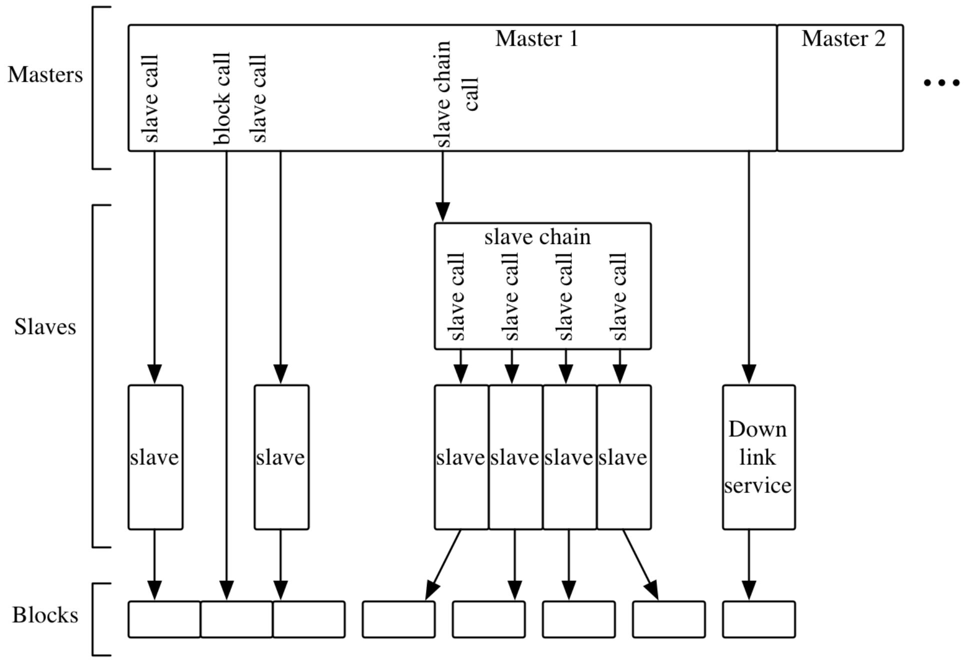

Figure 7: Master / slave architecture with onboard blocks for commanding to reduce uplink volume

B. Phoenix entry, descent, and landing on Mars

The successful 2009 landing of Phoenix on Mars[10,11] required correctly implementing a very challenging mission phase: Entry, Descent, and Landing. Responsibility for EDL activities was divided between attitude control flight code for high-rate monitoring and actuation, and VML blocks for everything else.

The approach taken within the VML sequences was state-driven. A series of 24 blocks, as a group, composed the mainline set of EDL activities. The mainline blocks made use of both timed and event-driven sequencing, using programmed delays for times prior to atmospheric entry, and taking events from flight software for activities starting with parachute deployment and ending with touchdown. This allowed the reusable blocks to be shifted relative to ground observations of the Martian atmosphere. It also allowed tightly timed activities to be initiated by physically unpredictable events, such as when the spacecraft would be descending slowly enough to deploy the parachute, at the correct speed to release the heatshield, or at a low enough altitude to release the backshell and begin powered descent. The state-driven blocks used under VML 2.0 to accomplish EDL became the inspiration to create executable state machines in VML 3.0.

One feature this state-driven approach enabled was a rapid reconfiguration of the EDL activities due to pre-launch debris analysis. Originally, the cruise stage (which contained all of the direct-to-earth communications capabilities) was to be jettisoned in preparation for entry after the spacecraft had slewed to its proper entry attitude. At the time this seemed sensible, as the success of the slew could be quickly verified via telemetry before the X-band transmitters were lost in the subsequent separation. However, well after the EDL sequences had been completed and EDL testing had been undertaken, analysis showed the risk of debris impacting the spacecraft due to the disintegration of the cruise stage to be too high. The slew-then-separate ordering would have subjected the lander to a high risk of severe damage from fragments. Instead, separate-then-slew was needed.

Recoding of the EDL sequencing was dramatically simplified by the state-based design. The state responsible for slewing was reordered to occur after the state responsible for the cruise stage separation. Unit tests were rerun, and the results verified. The new product was ready for use on the spacecraft in a matter of hours rather than the weeks it would have taken without VML, demonstrating the power of the VML state-based approach.

C. Mars Reconnaissance Orbiter

MRO employed a science sequencing solution using multiple parallel threads of execution. In order to decouple most of the science planning from the engineering sequencing, each instrument team was assigned a sequence thread or “engine” of its own and a block library of its own. Provided they stayed within predetermined timeframes and spacecraft states, they could command their instrument without coordinating with the rest of the system. This operability tactic limited the extent of any particular instrument command error. Likewise, corrective actions such as block updates only needed to be loaded on that instrument's engine.

D. Fault detection, response, and recovery

One of the more complicated aspects of spacecraft missions is to detect faults, undertake some appropriate response (if necessary), and recover the spacecraft to operational status after a fault has occurred. Due to their flexibility and ease of modification, VML blocks are frequently used to implement fault protection responses aboard the spacecraft. Upon observing sufficiently severe faults, high-level fault protection (HLFP) flight software may need to cause the spacecraft to enter safe mode or take some other action. In this case, the HLFP system stops all activities in sequence engines and load its own blocks to implement needed safe mode activities such as maintaining a power-positive attitude, initiating communications, or adjusting thermal settings. By coding fault protection responses as VML blocks, the responses are easily updated as the mission proceeds and the aging spacecraft develops idiosyncrasies.

Recovery may involve a large number of steps to take the spacecraft out of fault protection attitude, increase communications rates, download recorded data, etc. Blocks that automate portions of the process eliminate light speed delays inherent in most deep space missions and speed recovery. Once the ground determines the efficacy of each step, the recovery process can proceed rapidly.

VI. Testing

An important operability consideration is how pre-launch testing is accomplished. JPL has a test-as-you-fly policy, bringing operational tools into the flight system integration and test process early in Phase D. At the same time, operations teams are constituted and training begun. Where possible, operators assist with hardware test, and hardware developers assist with development of procedures, conduct training, and sometimes continue into operations.

Early operator involvement and coordination during testing can lead to benefits for both operators and testers. Operators learn the development history and get hands-on training, and hardware testers gain insight into how the item will be used. This cross training improves adaptation, test scripting, and future design. Another way VML missions benefit is found in testing of blocks and flight rules. When used in a hardware integration test, blocks and flight rule checks can be verified on the hardware as well as in the ground simulation. Minor flight software changes can be suggested at this point that could obviate the need for time-consuming workarounds in flight.

A. Mars Reconnaissance Orbiter: moving the spacecraft during ATLO

VML's ability to tap into the telemetry data of the spacecraft flight software proved to be very useful during testing of the Mars Reconnaissance Orbiter. During ATLO, a spacecraft has to be moved between facilities - from the high bay area to vibrational testing and back, then out to large thermal-vacuum chambers and back, and finally into a shipping container and onto a truck for the cross-country trip to the launch facilities. This process can be very stressful on completed work, in particular on wiring. After each move, a "touch test" is performed wherein every switch has to be cycled on and known results such as current readings, voltage changes, or temperature increases verified in telemetry, then cycled off with a similar check. When performed manually with ground-dispatched commands and human-verified telemetry readings on screens, the post-move touch test for a spacecraft like MRO could take almost a full working shift to complete, with all the associated personnel and facility costs.

In an effort to reduce the cost of performing the post-move touch test, a test conductor on the MRO mission used VML to issue the commands and perform checking of telemetry items to verify that the switches were in working order. By removing human interaction from the test once the blocks were loaded and execution begun, the automated VML touch test could be completed in minutes rather than an entire shift. Accuracy was perfect as well, as there could be no misinterpretation of screen data or missed human-dispatched commands. The test was sufficiently quick that it was performed on a more frequent basis, sometimes under circumstances other than after a move. In at least one instance, the automated touch test correctly identified damaged wiring, quickly allowing the spacecraft to be repaired and activities to be resumed.

B. Mars Odyssey

As the first spacecraft to employ the capabilities made available under VML 1.0, and having launched in 2001, Mars Odyssey has had the most experience with engineering blocks used to perform complex spacecraft activities. This experience brought a lesson about parallel vs. in-line invocation from master sequences to blocks.

One example learned in test was whether to spawn a Deep Space Network contact block that ran during an entire contact pass, or to run separate start contact and stop contact blocks. The former required running on a parallel, dedicated engine, preprogramming the block with the contact times via parameters. The latter involved simply invoking two separate blocks from the master sequence at appropriate times. Odyssey chose to implement the single block design, which in turn required the master sequence generator to account for the contact time in the invocation and the time for the activity following the contact, as well as allocating another engine for the parallel execution. The complexity of the generation and the dedication of an extra engine to run during the contact was noted during test, but remained during the mission.

Subsequent missions replaced the single contact block with separate blocks to start and end the contact. This approach had the advantage of allowing the block to be executed as a call from the master sequence, eliminating the need for using a dedicated engine or coordinating activities with block completions. By calling from the master for the start, activities could be naturally interleaved within the master, followed by the end contact block as just another activity. The start/end approach fit more consistently into the master sequence than did the parallel execution case, and used fewer resources.

C. Automated testing faster than real-time

One of the advantages of VML is its ability to run on a workstation. In order to test blocks and sequences, operators use Offline Virtual Machine, a program that combines the VML Flight Component with a command-line user interface. OLVM features a harnessed clock, allowing two extremes: time can be incremented far faster than real-time (up to 100,000 times real-time, as measured for Mars Odyssey), and time can be held still while users set up needed condition values. The users act in lieu of a master sequence, invoking blocks interactively with desired parameter values, changing global variable values, advancing time, and checking the results. The output can be captured and then played back into the system to recover keystrokes and drive the test, allowing large suites of unit tests to be built up for all operations products.

There are three advantages to this form of testing over the use of a mission Software Test Laboratory (STL) containing flight-like processors and full simulation. First, the testing is much faster, and runs on commonly available workstations, thereby allowing the user to iterate on the VML coding to correct errors with virtually instantaneous turnaround time. This allows the sequences to be fully correct well before needing to test these products in the expensive and tightly scheduled real-time environment of the STL. Second, the ability to automate testing of the products allows changes to be rapidly and thoroughly retested, with minimal effort on the part of the developer. Third, some master sequences cover weeks of time, making STL testing impractical or impossible, thereby requiring a much-faster-than-real-time approach to allow the product to be tested on the ground before installation on the spacecraft. VML provides convenient mechanisms to enable users to complete sequence testing as quickly and efficiently as possible.

VII. Adaptation

Adaptation is the process whereby the basic VML processing setup is customized to meet the needs of a specific mission. As the flight code evolves through the development phase, the ground tools evolve in synch. All VML missions benefit from each other's lessons working with the system and the tried-and-true adaptation and operations practices developed over 15 missions. This working community allows each mission access to methods refined by previous missions while maintaining their independent adaptation strategy.

A. Layered architecture for easy flight software adaptation

VML is built in a layered architecture to easily adapt to new missions without the need for any legacy software changes, and only minimal additions. One prime example of this is the flight software integration of VML into the existing flight code base performed for the Dawn mission.

The heritage sequencing capability in the flight software for this Orbital Sciences Corporation (OSC) mission was deemed inadequate for long-term maneuvering with a low-thrust ion engine. The VML Flight Component was installed to enable the mission to perform maneuvers and related activities using blocks developed by JPL. A small supervisory task controlled the interaction between VML and the OSC legacy code, incrementing the VML flight component discrete time clock, and passing command messages in from the software bus. Minor adaptation routines were emplaced to allow VML access to the real-time operating system, telemetry reporting system, and command subsystem, the latter requiring stripping of the VML command opcodes prior to dispatch via CCSDS messaging. Since the OSC flight code base lacked any sort of file system, a file buffering system was also written to allow files to be placed in known locations for loading as though a file system were present.

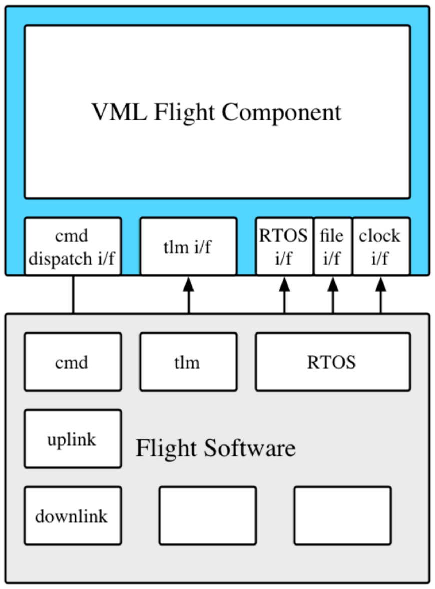

Figure 8: Typical layered architecture code adaptation between VML flight component and legacy flight code

The basic arrangement for adapting VML to an existing code base is shown in Figure 8. Note the intercommunication between command dispatch and commanding software, a telemetry interface, and interfaces for real-time operating system routines, file access, and clock reading, which constitute the majority of needed interactions. Due to the layered architecture, VML works in the same fashion when installed into any legacy software systems as it does for the as-flown Lockheed-Martin and Orbital Sciences missions.

B. Instant Adaptation

VML allows a set of known processes and products to be instantly specified when a mission choses it. A basic adaptation of VML can be installed in the ground system within a few hours. Flight and ground software developers can immediately begin working with the system, identifying interfaces and even developing sequences. By providing a readily-available baseline system, developers can more quickly concentrate on the unique aspects of the system that require more attention, rather than spending time and resources recreating typical capabilities.

C. Cloning

A more complex adaptation can be developed by selectively cloning a similar previous mission down to the commands and telemetry. For instance, missions using Lockheed-Martin spacecraft buses have a highly refined primary adaptation that evolves from mission to mission to accommodate evolution of their spacecraft product line.

D. Product reuse

In the same way that cloning a similar mission's adaptation can speed early adaptation, so can reuse of well-tested blocks. For similar missions, blocks can be copied and then adjusted for the new mission via minor tweaks. As the experience base with VML has grown, a collection of multi-mission blocks has grown as well. Methods for commanding difficult activities have been noted and advice for future users documented. Because VML is multi-mission by its nature, its users can share knowledge on missions with little or no commonality, e.g. an EDL mission vs. Spitzer, or missions built by different spacecraft developers.

VIII. Looking Ahead: What's Next

The future of flight/ground integration is bright when using VML. VML capabilities have been enhanced in order to get missions started quickly, improve ease of configuration, reduce the embedded memory footprint, expand capabilities, and simplify implementation of operational systems. Mechanisms for these improvements include human-readable ASCII commands, VML-specific standardized configuration tools, and the ability for operators to create expert systems built from state machines.

A. ASCII commands and reduced embedded memory footprint

The traditional JPL process is to use a complex database and tool set to produce all combinations of commands with their parameters, and track each individual command as a binary form ready for uplink. As the database is updated, new command files with incremented revision numbers are produced, maintaining the old command revisions for use with older flight software builds. The result is a large number of files, typically numbering in the hundreds of thousands, with the potential to radiate out-of-date commands to the spacecraft, or to fail to update testing scripts to use the latest version of commands. VML ASCII commands implement a less complex mechanism for building and managing spacecraft commands, providing distinct advantages over ground-built binaries. A comparison between ground-built binaries and VML-translated ASCII commands is shown in Figure 9.

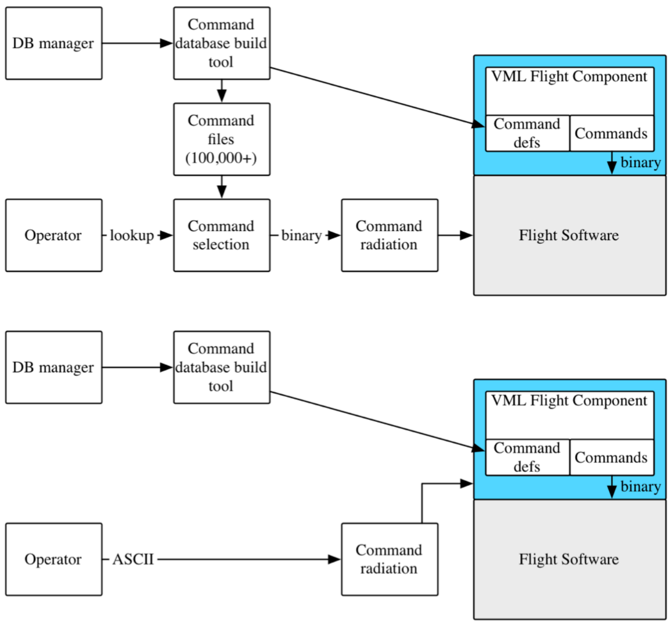

Figure 9: Traditional JPL binary command flow large combinations of parameters and revision tags in uplinkable binary files vs. simplified VML 3.0 ASCII command flow and onboard command building

Since the VML flight component is required to dynamically build commands onboard the spacecraft, it contains a complete representation of all valid commands, including names, opcodes, data sizing, ordering, parameter ranges, state translations, etc. The VML 3.0 command database and data translation tools produce the command representation map compiled into the VML flight component. All the ground need do is radiate the human-readable ASCII version of the command to the VML flight component, which will validate and translate the command using its onboard command definitions, and pass the resulting binary on to the flight software for dispatch and processing. Ground operations testing scripts need only call out the actual ASCII representation of the command without worrying about revisions, since the spacecraft will always have the ability to translate the command into a binary form compatible with itself. This eliminates the non-value-added effort of updating scripts when database releases occur, and the risks associated with missing such an update.

Another advantage to the ASCII command approach is that only a rudimentary commanding capability is needed, which can be as simple as using a command line capture program that forwards the resulting characters over a Unix TCP/IP socket to the uplink flight software. Simplicity is particularly useful early in the software development cycle before elaborate operations environments are available, or for low-budget technology demonstrations intended only for the lab. The ASCII commanding also can be embedded directly in user interface elements, including MatLab controls. Several projects in development are currently using this capability, including Resource Prospector Mission, AutoNav development, and the NASA SBIR phase 2 I-SPAREX project[13].

B. Reactive sequencing and state machine expert systems

The response of a sequence to environmental conditions that cannot be predicted via time is called *reactive sequencing*. The technique was first created for the Spitzer Space Telescope in order to use the settled state of the observatory as a precondition to imaging after a slew. The technique was also used extensively during Phoenix EDL, which needed to wait on conditions for parachute deployment, heat shield jettison, backshell separation, and touchdown.

VML state machines are the culmination of reactive sequencing, using variable values as preconditions to taking transitions between states. State machines can't be considered time-ordered sequences: they are reactive logic constructs capable of autonomous decision-making within a well-defined domain. Placing autonomy into the sequencing domain rather than into the flight software domain makes the behavior of the resulting expert system visible. It also simplifies changing the expert system: merely by placing new files onboard with new state machine definitions, the system can be updated to reflect changes in the environment or the spacecraft. *If needed, entirely new autonomous capabilities can be incorporated into the spacecraft without changing any flight software.*

To date, the most extensive expert systems implemented in VML state machines have been related to spacecraft navigation controlling and sequencing JPL's AutoNav[7] software. These applications include autonomous comet / asteroid touch-and-go technical demonstrations[12], lunar landing simulation, halo orbit emulation with hardware in the loop, and autonomous rendezvous and docking for a Mars sample return mission[16]. Additional applications using state machines include coordinating complex instrument inter-activities on the Resource Prospector Mission, surveying for water on the moon and demonstrating oxygen production from regolith[15].

C. Design pattern flexibility

The design pattern for autonomous onboard operations, in particular, must stay flexible. VML 3.0 provides the fundamental capabilities for supporting many kinds of autonomous expert systems with coordinated state machines that work together as an ensemble to run elements of the mission. Coupled with other VML capabilities such as matrix math operations, logic constructs, insight into on-board telemetry values, and ground interaction via global variable values, VML offers an ideal environment for adapting existing implementations to new mission needs.

IX. Conclusion

Operability has been an ongoing problem in mission development. As systems become more complex, operating them has also become more complex. Making appropriate choices about mission capabilities can mean the difference between an operable system and one that requires constant attention. In an era of tight budgets and reduced operations staff, operable systems become more than important - they are critical. VML is a flexible, customizable solution to a number of operations challenges, and allows operations work to begin at mission inception, when operability considerations can have the greatest impact.

Acknowledgments

Some of the work described in this paper was carried out by Blue Sun Enterprises, Inc., under an agreement with the National Aeronautics and Space Administration, and administered by the Office of Chief Technologist as a Small Business Innovation Research grant.

Some of the work described in this paper was performed at the Jet Propulsion Laboratory (JPL), managed by the California Institute of Technology (Caltech), under contract to the National Aeronautics and Space Administration (NASA).

References

Reports, Theses, and Individual Papers

[1]Grasso, C. A., Lock, P. d., “VML Sequencing: Growing Capabilities over Multiple Missions”, AIAA Space Operations Conference Proceedings, April 2008.

[2]Grasso, C. A., “The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)”, IEEE Aerospace Applications Conference Proceedings, March 2002.

[3]Grasso, C. A., “Techniques for Simplifying Operations Using VML (Virtual Machine Language) Sequencing on Mars Odyssey and SIRTF”, IEEE Aerospace Applications Conference Proceedings, March 2003.

[4]Peer, S. and Grasso, C. A., “Spitzer Space Telescope Use of Virtual Machine Language”, IEEE Aerospace Conference Proceedings, December 2004.

[5]Grasso, C. A., “Virtual Machine Language (VML)”, NPO 40365, JPL Commercial Programs Office, Innovative Technology Asset Management Group, Docket Date: 12-May-2003.

[6]Grasso, C. A., “Virtual Machine Language (VML) NASA Board Award”, NASA Inventions and Contributions Board, NASA Technical Report 40365, Award Date: September 7, 2006.

[7]Riedel, J. A., et al., “AutoNav Mark 3: Engineering the Next Generation of Autonomous Onboard Navigation and Guidance”, AIAA Guidance, Navigation, and Control Conference, August 2006.

[8]Chapel, J. et al., “Aerobraking Safing Approach for 2001 Mars Odyssey”, American Astronautics Society Guidance and Control Conference, Feb 2002.

[9]Grasso, C. A., Riedel, J. E., “VML 3.0 Reactive Sequencing Objects and Matrix Math Operations for Attitude Profiling”, AIAA Space Operations Conference Proceedings, May 2012.

[10]Grover, M., Cichy, D., Dasai, P.N., “Overview of the Phoenix Entry, Descent and Landing System Architecture,” AIAA Paper AIAA 2006-7218, AIAA/AAS Astrodynamics Specialist Conference, Honolulu, HI, 18-21 August 2008.

[11]Garcia, M., Fujii, K., “Mission Design Overview for the Phoenix Mars Scout Mission,” AAS Paper 07-247, AIAA/AAS Space Flight Mechanics Meeting, Sedona, AZ, 28 January -01 February 2007.

[12]Grasso, C. A., Riedel, J. E., Vaughn, A.T., “Reactive Sequencing for Autonomous Navigation Evolving from Phoenix Entry, Descent, and Landing”, AIAA Space Operations Conference Proceedings, April 2010.

[13]“Balancing Autonomous Spacecraft Activity Control with an Integrated Scheduler- Planner And Reactive Executive (I-SPAREX)”, Red Canyon Software, SBIR Proposal #: H6.01-8798, NASA contract NNX13CA26P.

[14]Grasso, C. A., “Formal Methods for Design, Development, and Runtime: Runtime Verification of Distributed Reactive Systems Using DR-VIA and RTV with extended TTM/RTTL Notation.” Doctoral Thesis, University of Colorado, 1996.

[15]Office of the Chief Technologist, National Aeronautics and Space Administration, “Virtual Machine Language Controls Remote Devices.” Spinoff 2013, http://spinoff.nasa.gov/Spinoff2013/pdf/Spinoff2013.pdf.

[16]Grasso, C. A., “VML 3.0 Reactive Rendezvous and Docking Sequencer for Mars Sample Return”, AIAA Space Operations Conference Proceedings, May 2014.

Related web sites

Blue Sun Enterprises VML Website http://www.bluesunenterprises.com