VML Sequencing: Growing Capabilities over Multiple Missions

Dr. Christopher A. Grasso *
Jet Propulsion Laboratory / California Institute of Technology / Blue Sun Enterprises, Boulder, Colorado, 80302

and

Patricia d. Lock†
Jet Propulsion Laboratory / California Institute of Technology, Pasadena, California, 91109

1. Introduction

V
irtual Machine Language (VML) is an award-winning advanced procedural sequencing language in use on NASA deep-space missions since 1997. Missions featuring VML include Spitzer Space Telescope, Mars Odyssey, Stardust, Genesis, Mars Reconnaissance Orbiter, Phoenix, Dawn and Juno. The latest deployed version, VML 2.0, features parameterized functions, conditionals, polymorphism, a rich set of control directives and data types, event detection and response, and on-the-fly creation of spacecraft commands. This feature set is used to simplify spacecraft operations and science gathering activities. A new 2.1 version is being prototyped for use as an executive within flight instruments, and may be deployed on Juno.

VML is used for a diverse set of mission functions on its various host spacecraft, including launch sequencing, daily activity loads, orbit insertion, aerobraking, entry-descent-landing, science observation, and fault responses. On Dawn, VML is used to autonomously control thrust output of the Ion Propulsion System. Generic implementations of several major uses are presented. Functional problem factoring and resource utilization are also considered.

VML is divided into three major components. The flight component exists onboard the spacecraft, allowing VML sequences to run within the flight context. The VML compiler translates human readable sequences into binary executables placed onboard and loaded by the flight component. Offline Virtual Machine is a workstation program that marries the flight component to a user interface, can run sequences at several hundred thousand times real-time, and provides a runtime behavior with 100% fidelity to the flight context. Each of these components is used in the development and deployment of sequences for flight. This paper discusses the use of these components in typical operations development processes on missions like Mars Odyssey, Phoenix, and Dawn.

Blocks are reusable relative time-tagged sequences that parameterize routine operations, and are typically packaged together into single uplinkable files called libraries. Sequences are single-use sets of instructions that run in absolute or relative time. The relationship between reusable blocks and one-use sequences is discussed. Reduced development effort due to iterative block development is outlined, along with typical development procedures. The lower cost and reduced complexity involved with creating blocks rather than flight software is noted, as is the reduction in uplink size. The ability to migrate to the spacecraft functionality that is more traditionally implemented on the ground is examined. The implications for implementing spacecraft autonomy without the need for expensive flight software agent development are discussed.

Increasingly more capable versions of VML have flown on a series of missions. The arc of VML 0, VML 1.0, VML 1.1, VML 2, and VML 2.1 is examined. Given VML's long lineage of missions and increasing capability, further simplification of operations using features in VML 2.1 is discussed. Finally, the application of lessons learned on each of the VML missions, and the incorporation of new features based on these lessons, is provided. VML is available for distribution free of charge by the Jet Propulsion Laboratory under NASA Technical Report 40365.

* Principle Flight Software Engineer, Blue Sun Enterprises, 1942 Broadway Street Suite 314, Boulder, CO, 80302, Professional Member.
† Senior System Engineer, End-to-End Information System Engineering, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA, 91109, Member.
II. Development Backdrop

A. The Olden Days: One-offs and Cul-de-sacs

Back in the time of large, unfriendly computers, commanding a spacecraft was a large and cumbersome process. One set of tools was needed to design science observations and another set created the commands to carry out those designs. Separately, flight software was developed to operate each spacecraft. Since each spacecraft was unique, its flight software was also unique. In some cases, flight software could be reprogrammed in flight, but it was an esoteric and risky process, usually reserved for extreme cases such as when the Galileo mission's high gain antenna became stuck in a partially opened position.

Ground software and flight software were developed in parallel pre-launch but the development processes were decoupled, with separate teams, sets of requirements, and mindsets. Ground had no influence on flight, and flight had a tendency to overlook operations needs. The assembly and test phase of development often used a third set of tools until close to launch, leaving operators with little or no hands-on experience in using the flight system they would be responsible for keeping safe.

Voyager, Galileo, Cassini, Mars Observer, and Magellan all were developed under this paradigm, with small advances from mission to mission. What heritage could be maintained was transmitted by personnel rather than by software reuse, and therefore was subject to loss by reassignment or retirement of personnel. In operations, those who moved among missions brought with them an understanding of what could be the same and what must be different from spacecraft to spacecraft. A consensus grew that rather than only bringing personnel experience forward, tools and systems should also be reused. During this period, space exploration budgets became tighter as well, and a shift from large flagship missions to smaller more agile missions began.

B. Sea Change: The Multimission Mindset

With the advent of the Pathfinder mission, JPL developed a wide range of new technologies for spaceflight: a base station / rover combination, airbags, use of a radiation-hardened processor similar to commercial PowerPC chips, and modularized flight code intended for easy reuse on future missions. Far from creating a one-off implementation, Pathfinder showed the multimission mindset taking hold.

The Pathfinder lander software architecture (excluding the rover), in particular, would have considerable impact on the Stardust, Mars Climate Orbiter, and Mars Polar Lander missions that featured VML 0. First, rather than featuring custom software from the lowest levels up, Pathfinder's flight software was built on a space-rated version of a modern real-time operating system, VxWorks. This provided convenient scheduling of multiple tasks, mutual exclusion constructs, a concept of priority, and a commercially supported development environment. The flight code featured more sophisticated telemetry than the typical subcommutated frame map, instead using event reports for rapid time-tagged reporting of conditions, and a channelized "push" telemetry system to allow tasks to cyclically place telemetry information into a separate telemetry reporting task. The opcode / parameter commanding was mapped into a messaging protocol. The features of messaging, event reports, channelized telemetry, and task scheduling would all find their way into VML flight code.

Despite its design for reuse, the Pathfinder code base provided only a rudimentary sequencing capability. Sequences featuring conditional checks and other sophisticated activities were implemented in C code as single use flight software modules and uplinked to the spacecraft on a daily basis. Once onboard, they were loaded into the flight software image space via VxWorks call, effectively relinking the flight software into one or more new configurations each mission day. The sequencer consisted of separate VxWorks tasks, each of which would wait until a designated time before calling a routine in one of the newly loaded flight software modules. Fundamentally, this implementation was a set of sequence lists, but had the side effect of requiring a new flight software configuration for each sequence. One downside of this approach was the difficulty of testing the sequence software loads using a faster-than-realtime virtual clock in a non-VxWorks environment. A second downside was the exposure of the full flight software load to any mistakes made in the C coded sequences. A final downside was that this approach propagated the lack of coordination between flight and ground systems prevalent in preceding missions.
C. Clean Sheet

Spitzer (originally named SIRT F, the Space Infrared Telescope Facility), the last of the four Great Observatories, was a new kind of mission for JPL. Rather than time-constrained mapping or flyby observations, Spitzer could observe a large portion of the sky for several weeks, repeating on 6-month cycles. Spitzer's cryogen use strategy also constrained operations to using one instrument for one week, then switching to another instrument for the next week. A time allocation committee chose science observations from among peer-reviewed proposals. The selected observations were entered into a database of potential observations, each observation having a validity window rather than a specific start time.

A commanding strategy was explored that would load the observation specifications into onboard tables. The spacecraft flight software would choose at runtime which observation to execute next, based on duration, remaining time in the observation window, time left before downlink, etc. Because spacecraft turn duration was non-deterministic, it was believed that this approach would also increase observation schedule efficiency by allowing the spacecraft to control start times based on local knowledge of turn completion, rather than by adding slew margin on the ground. This in turn would allow more data to be taken over the life of the mission before the cryogen ran out.

As this strategy was analyzed, it became clear that none of the onboard sequencing applications currently in use could perform this type of commanding, nor could any of them be readily adapted to the table structure. A new flight software sequencing application was needed for Spitzer. This new flight software would be expensive, risky and likely not very adaptable to planetary missions. Based on schedule and budget constraints, another solution needed to be found: Virtual Machine Language sequencing.

III. What's a VM?

A. Like a Little Computer: Emulated Processing

The basic programming approach familiar to most spacecraft operations personnel is procedural in nature. Named routines stored in an instruction space call other named routines, passing parameters and receiving results. Data is stored in a separate memory space, and is altered according to defined arithmetic and logical rules. At any instant in time, there is one instruction being executed. There is a concept of control flow: one instruction is considered as "next", either because it immediately follows the current instruction, or because branching logic forces execution to jump to some other instruction. The entire conceptual framework of the modern computer, with instructions, data, a current position, and branching control flow can run either directly on a central processing unit (CPU), or within a software-simulated processor called a virtual machine. VML takes the latter approach.

The virtual machine, or VM, is implemented as a complex data construct acted upon by flight software. A simplified form of this data construct is shown in Figure 1. The VML flight software interprets instructions in the virtual machine, using an index to track the current instruction number much like a hardware CPU tracks the address of the current machine instruction. When branching instructions are encountered, the software interprets whether to take the branch or to continue to the next instruction, and changes the value in the instruction index to reflect the decision made.

The operand stack is used to store variables found in various functions, parameter values for calls to functions, and instruction indices for returning from function calls. Simple data types like integers and floating point values are compact enough to be stored directly on the operand stack. For efficiency, string operands use separate string storage space. Strings are typically much larger than integers or floating point values, and vary widely in the number of bytes required to hold a value. Specialized string storage allows bytes to be packed together and referenced by operands in the operand stack. The string storage is implemented as a separate data space in the virtual machine construct.
Spacecraft commands, like strings, also vary a great deal in length. A NOOP (no-operation) spacecraft command might only take one or two bytes, whereas some specialized instrument commands may take hundreds. Therefore, a specialized area is provided for command storage in order to avoid wasting memory.

The entire approach of instructions with a current instruction index, operands pushed onto and removed from the operand stack, and branching control flow allows the VM to emulate the kind of processing normally found in computing hardware. The interpretation step allows the instruction set of a VM function to be specifically limited to high-level, sophisticated directives tuned to operational needs.

Traditional sequencing, by contrast, typically uses a list or a table of instructions with time tags indicating the time of execution for each entry in the table. Figure 2 shows this rudimentary capability. In a list, the instructions are executed in order, but feature no parallelism. In the sequence table, entries execute in parallel, each waiting for its time to come due.

The instructions invoked from the list or table might be spacecraft commands with fixed parameter values, or high-level directives invoked with fixed parameter sets. Any high-level directives would have to be coded in flight software in order to feature logic and branching, perform calculations, or check for conditions. This dependence on flight software removes the possibility of operations personnel implementing new conditional statements, loops, or calculations as part of the in-flight products, decreasing sequence flexibility and increasing costs.

The sequence table also features a lack of structure and coherence. Since any instruction may come due at any time, there is no sense of "current" and "next" that is fundamental to programming. This lack of structure can impede easy understanding of the intended functionality of any particular sequence load, as time tagged instructions with the same time of execution may exist anywhere in a comparatively large space of hundreds or thousands of entries. In addition, a lack of structure inhibits detecting events and responding to conditions, as we will discuss in the next section.

B. Less Is More: Being Next

The concept of control flow through code running on a CPU inherently implies a current instruction and a next instruction. When the current instruction finishes executing, the CPU must calculate which instruction in memory to execute next. Most of the time, the next instruction resides in the next memory address. Occasionally, a decision must be made regarding whether to execute the next instruction or to jump to some other instruction in memory. VML employs similar mechanisms to those used by CPUs.

Flow control in VML may take the form of simply continuing to the next instruction, performing a conditional check followed by a branch to a different location, calling a subroutine and returning, or starting execution of a new thread to run in parallel with the current thread. VML implements conditional checks using expression evaluation, then branches to a label in a manner similar to an assembly program. Calling preserves return information on the VM operand stack and uses the first instruction of a named routine as the branch destination, taking the return off the stack when the end of the routine is reached. Spawning starts a separate VM to take the first instruction of a named routine as its first instruction and runs in parallel with the spawner in a fashion similar to multitasking. In each case, the key to the structure of the function is to designate what the virtual machine is to do next.

C. Lots of Little Computers: Multiple Threads of Execution

With the basic execution paradigm defined, it is a simple matter to generalize from having a single virtual machine to having an arbitrary number of them. Each engine has its own instruction and operand space, and each engine provides one thread of execution. Due to the data-driven nature of the sequencing engines, VML can support
as few as one engine, or (accounting for hidden engines used for global variables and spawning) as many as 65,532, although this maximum configuration would probably entail a prohibitively large memory footprint.

The first VML 0 missions were sized to eight engines, mirroring the sequence capabilities VML replaced (see section IV B below). Odyssey and Genesis, using VML 1.0, also used eight engines, but due to its operational complexity, the Spitzer mission opted for twelve engines. VML 2.0 missions chose to further expand the number of engines: Dawn and Phoenix both run with 16 engines, while MRO uses 20 in order to more simply manage the large number of instruments onboard. Juno is exploring using 28 engines.

D. Tool Chain: Components Working in Concert

The VML tool suite allows the generation of files containing functions, the loading and execution of sequences, and the testing of sets of sequences. The relationship of each of these VML tools is shown in Figure 3. A source file containing human-readable VML script is generated using a standard editor or a ground data system tool. The VML Compiler translates a text file, or set of merged text files, into a loadable binary, translating spacecraft commands and absolute times using external mission-specific tools. The VML Compiler also has access to lists of valid global variables and symbolic constants for the mission. The module file produced can then be loaded by the VML Flight Component.

The typical development process involves running the compiled module under Offline VM (OLVM) in order to test and validate the behavior of the code. OLVM is capable of performing user-defined tests automatically by first capturing a user-guided session, then extracting user keystrokes from the human-readable session output and rerunning the test. This automates the testing process for very little investment of effort. OLVM can be widely deployed on relatively modest workstations, including Sun Ultra, Intel Linux, PPC Macintosh, and Intel Macintosh platforms. Developers can thoroughly test products before taking them to the more expensive, slower, and less available real-time software test lab. VML products tend to work the first time in the lab without further modification when their development features OLVM testing. In some cases, products span so much mission time (weeks to months) that a full run in a test lab is not practical or even possible.

E. VML: Human Readable Language

VML’s ultimate aim is to eliminate errors, yet provide enough power to simplify operational blocks and sequences. Fortunately, the two goals are complementary. By providing high-level constructs like conditionals and loops, discouraging branching with labels, encouraging functional abstraction via parameterized blocks, allowing graceful coercion and weak data typing, and building commands with calculated parameter values, VML allows concise, syntactically simple operations products to be implemented.

F. Reusable Blocks: Parameters, Conditionals, and Return Values

Perhaps the easiest way to explain VML scripting representations is by example. Figure 4 shows a code listing of VML with blocks used for setting up and stopping contacts between a notional mission and the Deep Space Network used for communicating with interplanetary missions. Blocks are intended to be reused, sometimes thousands of times. In this sample code, the complexities of choosing bit rate indices, actuating a solid-state power amplifier with timing appropriate for the safety of the electronics, and turning off the electronics are abstracted as the blocks. VML is not case sensitive, but for demonstration clarity, VML keywords have been fully capitalized.

The function convert_bit_rate is a reusable block that accepts one parameter. Notice that no data type specification is necessary, but may be optionally provided to cause a coercion if desired. The parameter is treated like a local variable, and may be used in expressions inside the block. Values may also be assigned to it within the block. An arbitrary number of parameters may be expected by a function, as well as no parameters at all. Any missing parameters in the call to the function are gracefully substituted with the special data type and value UNKNOWN. Any excess parameters in the call are ignored by the function.
Next comes the declaration section, which in this case features one declaration of a local variable `bit_rate_index`. The variable has an initialized value of 0. All variable declarations include an initialization value that is evaluated upon entry into the function. This eliminates nondeterministic behavior due to random initialization values, and prevents corrupt floating point values from causing exceptions on CPU hardware.

The body of the function contains the executable instructions, bounded by `BODY` and `END_BODY`. The instructions within a function execute according to timing relative to one another. In versions of VML prior to 2.1, explicit time tags were required on every statement. This proved cumbersome, so starting with VML 2.1, if no time tag is present, the VML compiler substitutes a time period of one tick. This has the effect of separating the execution between statements. Zero time tag values are allowed, in which case instructions are executed in the order encountered, but the next instruction after the current instruction is issued on the same tick as the current instruction.

Within the body of `convert_bit_rate` are instructions implementing a cascading conditional structure with three matching conditions. These work like ordinary `IF` statements in most computer languages such as C or Ada. There is no need for an explicit conditional body for multiple statements as in C: every statement between the `IF` and the eventual `ELSE_IF`, `ELSE`, and `END_IF` is automatically considered to be part of the body of the `IF`, eliminating a source of coding errors. The `IF` statement clauses shown here feature complex expressions, including in some cases Boolean logic operators like `AND (&&)`, `OR (||)`, `XOR (@@)`, and `NOT (!)`.

Values can be returned from functions one of two ways: `RETURN` statements (shown in the example), which provide a single value back to the calling statement that can be assigned within the context of the calling function, and `INPUT_OUTPUT` parameters. The latter are not shown in this example, but provide a reference to a variable in the caller's context rather than a copy of such a variable. Assignments to the `INPUT_OUTPUT` parameter change the original variable rather than a local copy. If a literal is given in the call corresponding to an `INPUT_OUTPUT` parameter, VML quietly treats the parameter as an `INPUT` parameter, making a local copy of the value. This eliminates any need to detect and respond to mismatch errors.

The next function, `dsn_contact_start`, accepts two parameters, one of which has been coerced to a string and checked for valid values. Failure to match a valid value for the mode parameter results in a runtime error. The function starts by calling the block `convert_bit_rate` discussed previously, receiving a value to use for the `bit_rate_index`. It then goes on to dispatch commands using `ISSUE` statements. Notice the `R0.2` time tag between the first two `ISSUES`: this forces a 0.2 second relative time delay between the completion of the first command and the dispatch of the second command. VML 2.1 has facilities for detecting the completion of commands, whereas previous versions only timed the dispatch of commands relative to the previous command dispatch. Since VML does

Figure 4. Sample blocks. These sample blocks for a notional mission would be placed together as a library. They provide simple abstractions for complex operations, and are used in place of individual commands.
not understand a spacecraft command per se, it depends on an external translation tool to provide the VML Compiler with the binary equivalent of whatever text is present to the end of the line after the `ISSUE` keyword. This allows enormous flexibility in spacecraft commands, and guarantees that VML syntax will never collide with spacecraft command syntax.

Next come two examples of dynamically issued commands, separated by a programmable delay. The dynamic commands are built by looking up command formats in a mission-specific data structure compiled into the VML flight component. Notice how the `bit_rate_index` value return from a previous block call is used in the second `ISSUE_DYNAMIC` statement. Command parameters are checked against the mission-specific command definition data at runtime for valid ranges and state values, with violations resulting in a command dispatch error that will abort execution if engine aborts are enabled. In the case of a violation, no command is actually dispatched.

Finally, at the end of the function, a global variable `gv_dsn_contact` is set to the logical value `TRUE`, which can be tested in other blocks (and even non-VML flight software) in order to affect runtime behavior.

The use of blocks reduces development effort, providing a high-level representation of complex functionality without imposing the kind of heavyweight review process required by flight software. Blocks, once developed, remain in most cases unchanged. Only the invocation of a block need be reviewed when examining the sequences that use it. This substantially reduces the amount of time required to review activities and reduces the risk of error.

G. Sequences: Activities Using Blocks

A second major kind of function is the sequence. Sequences are intended to be used exactly once, and come in two varieties, absolute and relative. Absolutely timed sequences contain absolute time expressed either in spacecraft time (seconds since a configurable epoch) or Earth-time, with the latter requiring a conversion tool. Absolute sequences may also contain relative times, and these will be offset from the previous statement. Relative time sequences contain only relative times similar to those seen in the example blocks above. This allows some flexibility in when the sequence is initiated, but causes the sequence to proceed according to precise timing.

Figure 5 shows an absolutely time tagged sequence `master_4`. This sequence automatically executes upon being loaded into an engine, and automatically vacates the entire file loaded into that engine when it is finished executing. This particular sequence loads a file containing a `slave` sequence that performs observations. Ten seconds after issuing the load command, the master sequence starts execution of the relative sequence `observe_day_39` (not shown here). The master sequence handles the loading, starting, and stopping of slave sequences, and the initiation of DSN contacts.

At a later time, the master sequence makes use of a `PAUSE` statement in order to suspend observations while a contact with the deep space network proceeds. The `PAUSE` simply defers execution of the engine intact so that it may be resumed at a later time. Notice the use of the blocks previously discussed. Once the DSN contact has completed, `RESUME` allows the paused slave sequence to proceed. The ellipses in this master sequence are provided to indicate that there are other statements not shown, perhaps more slave sequences loaded, paused, and resumed, along with more DSN contacts.

At the end of the sequence the `HALT` statement terminates any unfinished camera observations, and the engine containing the observation sequence is unloaded to free it for future use. The final statement of the master sequence loads the next master sequence. Since the command takes a round trip through the command subsystem, the VML
Flight Component has time to unload the file containing master_4 from engine 1 before loading the file containing master_5. This technique is known as chaining, and has been used extensively on missions since Mars Odyssey.

H. Details: Arithmetic, Loops

A wide variety of arithmetic operators and built-in functions is available, but has not been shown in the examples above. Arithmetic operators include addition (+), subtraction (−), multiplication (×), division (÷), modulo (%), and power (^). Bitwise operators include bit-and (∧), bit-or (∨), bit-exclusive-or (⊕), bit-invert (~), bit shift left (<<), and bit shift right (>>). Logical operators include logical-and (&&), logical-or (||), logical-exclusive-or (@), and logical-not (!). Comparison operators include equal (=), not equal (!=), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=). The usual operator precedences are enforced, along with parentheses.

Built-in functions simplify calculation. For numeric values, these include ABS(), SIN(), COS(), TAN(), ASIN(), ACOS(), and ATAN(). String lengths in characters are calculated by the LENGTH() built-in function. In addition, string operators are available to concatenate strings (+) and to split strings into substrings. Left split takes all characters to the left of a given character position, including that character (−|). Right split takes all characters to the right of a given character position, excluding the character (|−).

Loops featuring condition checking (WHILE) and iteration (FOR) are available. Simple examples of each are shown in Figure 6. The WHILE loop checks a logical condition in order to continue, whereas a FOR loop uses a local variable to count from a given starting value up to or down to a given ending value. The explicit designation of counting up or counting down makes the syntax of the FOR loop simpler than is found in the C language. The body of the WHILE loop is bounded by END_WHILE, and the body of the FOR loop is bounded by END_FOR.

I. Event Driven Sequencing

Event-driven sequencing provides a compact syntax for waiting on conditions, and proceeding when those conditions are met. This allows blocks to react to external signals from flight software and other blocks without an unduly complex implementation, in turn reducing both the cost of implementation and the risk of error.

Event detection takes the form of a variety of WAIT statements, shown in Figure 7. These instructions suspend execution of the running engine until a new value arrives, using no CPU processing: these are not spin locks, but instead are very efficient signaling mechanisms tied to global variable access routines. The simplest form of WAIT simply checks for a value to arrive, in which case the waiting engine is rescheduled. More complicated versions wait for changed values or wait for particular conditions to come true on the variable. In all cases, WAIT statements may assign the value in the global variable being waited on to a local variable, which can then be used in calculations. WAIT statements feature optional timeout values in order to guarantee that execution resumes if no acceptable value arrives within the specified time period. If the timeout expires, the local variable on the left side of the assignment is not assigned. This means that the user can definitively tell whether a figure.

<table>
<thead>
<tr>
<th>Figure 6. Loops. WHILE loops check logic. FOR loops iterate up or down by 1, or an optional step value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHILE gv_images &lt; 45 &amp;&amp; gv_camera_ready DO ... END WHILE</td>
</tr>
<tr>
<td>FOR i := 1 TO x DO ... END FOR</td>
</tr>
<tr>
<td>FOR i := gv_start DOWN_TO gv_end STEP 3 DO ... END FOR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Figure 7. The many-varied WAIT statement. Event detection takes the form of WAIT statements, which examine a single sequencing global variable. The WAIT can be simple, feature a comparison, or look for a change. WAIT statements contain optional timeouts in order to guarantee resumption of processing if the condition is not met.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK slew</td>
</tr>
<tr>
<td>INPUT ra</td>
</tr>
<tr>
<td>INPUT dec</td>
</tr>
<tr>
<td>DECLARE DOUBLE rates := 0.0</td>
</tr>
<tr>
<td>BODY</td>
</tr>
<tr>
<td>ISSUE_DYNAMIC &quot;SLEW&quot;, ra, dec</td>
</tr>
<tr>
<td>rates := WAIT gv_instrument_rate &lt; 0.001 TIMEOUT 10.0</td>
</tr>
<tr>
<td>rates := WAIT gv_instrument_rate</td>
</tr>
<tr>
<td>state := WAIT_CHANGE gv_state</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
value arrived or the statement timed out by first assigning an impossible value (e.g. -1), then checking to see if that value is present after the \texttt{WAIT}.

Simple semaphores are implemented using the \texttt{TEST AND SET} statement. This statement acts on a global variable, testing it as being greater than 0 and decrementing it all in one step before any other engine can execute a statement. If the value is zero or negative, execution of the engine is suspended until a non-zero value becomes present in the global, at which point the engine is again scheduled and attempts the \texttt{TEST AND SET} again. This capability allows global variables to guard mutually exclusive portions of functions to prevent miscommanding spacecraft elements, and allows classical producer / consumer code implementations as found in other computing environments. Like the \texttt{WAIT} statement, the \texttt{TEST AND SET} features an optional timeout in order to guarantee that execution is not permanently deadlocked.

\section{Flight Insight: Commands and Telemetry}

Controlling the activities of the VML Flight Component takes the form of commands dispatched to the software by the ground and by functions implemented by operators. These commands are straightforward, and provide a rich set of actions. They are listed below, along with basic descriptions. Note that some of the command names have changed slightly between versions: only the names found in the most current version of the VML flight software are given.

\begin{itemize}
\item \texttt{VM\_GV\_RENAME}: rename the variable at the given index with a new name
\item \texttt{VM\_GV\_SAVE}: save a range of global variables to a named file in the file system
\item \texttt{VM\_GV\_SET\_DBL}: set a global variable to a given double floating point value
\item \texttt{VM\_GV\_SET\_INT}: set a global variable to a given integer value
\item \texttt{VM\_GV\_SET\_STR}: set a global variable to a given string value
\item \texttt{VM\_GV\_SET\_TIME}: set a global variable to a given time value
\item \texttt{VM\_GV\_SET\_UINT}: set a global variable to a given unsigned integer value
\item \texttt{VM\_ABORT\_MODE}: set the engine to abort on an error (e.g. divide by 0, command error)
\item \texttt{VM\_HALT}: halt execution of the given engine
\item \texttt{VM\_HALT\_NAME}: halt execution of whichever engine is running the named function
\item \texttt{VM\_PAUSE}: pause execution of the given engine, allowing later resumption
\item \texttt{VM\_RESUME}: resume execution of the given paused engine
\item \texttt{VM\_LOAD}: load a file on a given engine, or choose an engine if a special "load to any engine" value is given
\item \texttt{VM\_LOAD\_SPAWN}: load the given file on an engine and spawn the given function with parameters in one step on that same engine
\item \texttt{VM\_SPAWN}: spawn the given function with parameters on a given engine
\item \texttt{VM\_START}: spawn the given function with no parameters on a given engine
\item \texttt{VM\_UNLOAD}: unload the given engine
\item \texttt{VM\_UNLOAD\_FILE}: unload whichever engine has the given file loaded on it
\item \texttt{VM\_PROTECT}: prevent an engine from being unloaded until it is unprotected
\item \texttt{VM\_UNPROTECT}: allow an engine to be unloaded
\item \texttt{VM\_RESERVE}: reserve an engine from being loaded unless its engine number is explicitly given
\item \texttt{VM\_UNRESERVE}: allow an engine to be loaded when the special "load to any engine" value is given
\end{itemize}

Monitoring the actions of an engine takes the form of telemetry channels. Each engine pushes its current function name, position, and file load to telemetry. In addition, the running state, load state, protection and reservation modes, and abort mode are reported. In order to monitor use, start count, start time, nominal termination count, and abort count are recorded. A variety of telemetry points also track the previous and next activity times and opcodes.

Taken together, the various features of VML provide a concise, powerful, standardized language for operating deep space missions. We will now examine the arc of missions over which VML has been used.
IV. Safe Sandbox

A. Spitzer Redux

When it became clear that the table-driven non-deterministic sequencing system (as discussed in Section II D above) was too risky and expensive for Spitzer, another solution had to be found. Coincidentally, VML was in the early stages of development for future missions. The Spitzer spacecraft developers co-opted VML and sped up its evolution to match the Spitzer schedule. In a break from previous missions, the system engineers from both flight and ground had to work together to define the set of capabilities that would fulfill Spitzer's needs, yet continue on the path of multimission reusable flight and ground software. This approach resulted in a feature set that would act as a flexible, standardized "front door" to the spacecraft for both Spitzer and future missions.

Spitzer's operational requirements stressed the existing flight and ground toolsets, and taxed the abilities of the small operations team. The Spitzer operations duty cycle was to observe for 11-1/2 hours, then turn and downlink for 30 minutes. Large data volumes and constrained onboard storage made it essential to get most of the data down in the first downlink opportunity. Because Spitzer's lifetime was limited by its cryogen, operating efficiently was essential to completing the mission's science requirements. Non-deterministic slew and settle durations also meant valuable observing time would be lost if worst case slew times had to be assumed in the planning of observations.

The earlier idea of letting the spacecraft choose which observation to perform next had been abandoned as too costly and risky. However, parts of that strategy could be applied through VML. In this case, VML was used to "pack 6 pounds of flour into a 5 pound bag" by intentionally oversubscribing the 11-1/2 hour observation window. Targets were chosen via a database on the ground and the observation sequence was assembled. Then, one or two extra observations were added to the end of the 11-1/2 hour window to take advantage of faster-than-expected slew and settle times. Global variables allowed the flight system to tell the sequence when the slew completed and settling was accomplished, and were used to trigger observation start. If an observation was never started or did not have time to complete, it was added back to the database and performed at another time.

VML made a major difference to Spitzer in the area of uplink volume. Sequences were uplinked to Spitzer only once per week. With only 30 minute DSN contacts, uplink volume was severely constrained. Much of Spitzer's observation strategy was highly repetitive. For some observations, the same command was issued over and over with only one or two parameters changing throughout the observation. VML blocks are ideal for this commanding style and in fact, Spitzer could not fit within its uplink limits without them. A study on the savings for Spitzer from using VML blocks revealed a reduction in uplink volume of 90%.

Using VML and multiple engines also allowed new fault protection strategies. Spitzer employed a master/slave sequence architecture. The master sequence ran on one engine and controlled the start times of science observations that were spawned onto other engines as slave sequences. The master sequence also controlled all absolute timed events such as DSN contacts and engineering activities. If a slave sequence ran into trouble or aborted, the master sequence was unaffected and could continue, ensuring that the spacecraft still made its next DSN contact. A sequence engine was also set aside for fault protection use. The fault protection engineers created VML blocks for their own purposes and spawned them onto the reserved engine to react to faults or to speed recovery from a faulted condition.

As Spitzer blocks were developed, the number of parameters in commands made holding all of the needed parameter combinations inside a block, and selecting the correct version using conditional statements, very large and unwieldy. Such commands needed to be built on the fly as the sequence executed. The concept of dynamic commanding was added to VML to address this issue. Originally, this dynamic commanding took the form of bits set within a local variable, but this approach also proved unwieldy, requiring large numbers of instructions to build a handful of commands. The final form of dynamically built commands incorporated specially made flight software and a data structure defining the format of every command in the mission. This approach allowed any command to be written without requiring prior planning, and allowed the same checks on valid command parameter values as were used on the ground. Dynamic command building found extensive use in instrument commanding, allowing the spacecraft to be configured to certain observation modes for large sets of images, conserving uplink and simplifying the sequence review process.
B. The VML Gambit: Stardust / MCO / MPL Sequence Software Lockup Fix

As part of the Stardust / MCO / MPL flight software development process, Pathfinder lander code was used as the development baseline for the sequencing capability. This activity proceeded in parallel with the VML development undertaken for Spitzer, but did not initially include VML code. Instead, the Pathfinder sequence code was to be enhanced with a comparison and branching capability. Implementation flaws in this code enhancement caused deadlock conditions among the eight sequencing tasks that manifested themselves during critical MPL entry, descent, and landing tests. The modified code was judged to be unsuitable for flight. Given the nature of the flaws and the relatively short period of time between failure manifestation and launch (approximately eight months), a whole-cloth emergency replacement of the sequencing flight code was deemed the lowest risk alternative available: VML would take an earlier flight than its originally scheduled Spitzer mission.

The partially implemented Spitzer VML 1.0 code was cut down to meet a reduced requirement set in order to speed production and deployment onto Stardust, MCO, and MPL. This version, referred to as VML 0, featured the VM engine core, simplified integer data types of global and local variables, basic arithmetic, bitwise operators, logical operators, and spawning of functions without any parameters. Approximately three months elapsed between the spacecraft operations testing failures and installation of the new VML 0 sequencing code, beating the launch date of the first mission by a scant five months. Experience on these missions would influence VML 1.0 development, providing flight time on the software and incorporating operations experience with the software base.

C. Odyssey: Spitzer's Other Testbed

After launch of Stardust, work resumed on the VML 1.0 flight code, and commenced on a second series of missions. Mars Odyssey, Genesis, and the Mars 2001 Lander were all ramping up production. While the Spitzer Space Telescope faced a variety of delays, operations teams were starting to identify the growing complexity of the Odyssey aerobraking maneuver around Mars. Aerobraking is a process whereby atmospheric drag is used to lower the apoapsis of a spacecraft, and requires precise timing and knowledge of orbital periapsis. A more capable sequencer than VML 0 would be necessary to lower risks and complete the aerobraking phase of the Odyssey mission. VML 1.0 provided a large set of data types, sophisticated block and parameter capabilities, loops, and conditional statements, making it the ideal candidate to control aerobraking. The decision to fly VML 1.0 on Odyssey first allowed aerobraking to be simplified, and later benefited the mapping mission. Spitzer also profited, as the early deployment on Odyssey identified enhancements needed before Spitzer's launch.

D. Genesis: Not So Fast

Despite being part of the next quartet of missions, all featuring the full VML 1.0 capability in flight software, mission managers on Genesis chose to continue with the limited ground system featured on Stardust. This decision was based on an effort to cap costs by eliminating changes from previous missions, coupled with the operationally simple nature of this solar sampling mission. Because of the decision, Genesis flew the binary translator originally developed for VML 0 due to the lack of the VML Compiler. The use of the translator limited Genesis' ability to use VML features. In order to maintain an identical flight software load across the four missions, the three full VML missions also flew the binary translator, although it remained unused. To simplify the code base and encourage operations personnel to share knowledge, the binary translator was eliminated from VML 2.0.

V. Missions, Missions Everywhere

A. VML 2.0: A Broader Audience

Early in 2001, the decision was made at JPL to target VML at more deep space missions across a larger range of contractors. The effort started with the VML 1.1 code base developed for Spitzer after the launch of Odyssey and featured uplink product size reduction features like string tables and time tag compression. New features included:

- a software abstraction layer for removing Lockheed-specific and VxWorks-specific service routines
- a new architectural layout to ease software integration
- a smaller memory footprint and incremental parsing of large files to distribute processor loading over time
- configurable alignment of data structures, configurable engine sizing, CCSDS compatibility
- for loops, trigonometric functions, compound expressions, optionally typed parameters
- removal of obsolete commands from VML 0 era, rename of commands, variable rename capability
- compatibility with Gnu tool chain, elimination of Solaris C compiler and non-standard compiler switches
- access to telemetry channels as read-only global variables, bringing more autonomy potential to missions
- reentrancy protection to prevent a block or sequence from running multiple copies simultaneously
These features helped attract new missions to the code base, including Mars Reconnaissance Orbiter (Lockheed) and Dawn (Orbital Sciences Corporation). Another mission using VML 2.0, the Hubble Robotic Vehicle mission, was canceled before phase C development.

B. MRO

Mars Reconnaissance Orbiter (MRO) brought a new set of challenges, and with it, a new set of VML functions. Its operational paradigm was to orbit Mars and allow the instruments to observe at will. Each instrument was allocated its own VM engine, with instrument teams taking responsibility for all instrument commanding. Separate virtual machines allowed partitioning of science activities from spacecraft engineering activities, reducing risk. MRO used 20 engines sized for the differing purposes.

MRO engineering activities included trajectory correction maneuvers that were controlled by blocks, using programmable delays and logic condition checking. As with Odyssey, blocks were used for aerobraking passes to maneuver the spacecraft to its final orbital altitude. MRO profited by another VML 2.0 enhancement: the ability to read and act on prechannelized telemetry. This allowed assembly and test engineers to run self-tests on the spacecraft after major movement and assembly events, reducing the risk of undetected problems and saving time, effort, and money.

C. Dawn

Dawn was a first for VML. Because of its heritage, VML had only been used on those missions with spacecraft developed by Lockheed-Martin, but the spacecraft contractor for Dawn was Orbital Sciences Corporation. Orbital's heritage software employed an old-school sequencing system that was incompatible with both Dawn's observing needs and with JPL's multi-mission ground system. A trade study found that it would cost less to enhance Orbital's heritage flight software with the VML flight code than it would cost to upgrade the onboard system or compensate for the incompatibility in the ground tools.

The VML 2.0 flight code was delivered to Orbital and integrated with their heritage system by means of a VM supervisor module. The supervisor allowed the VML code and the heritage flight software code to communicate with only a few changes to each set. Minor adaptations were also made to deal with the lack of an onboard file system and to add proper CCSDS headers to VM dispatched commands. The VML-Orbital collaboration was so successful that Orbital has requested permission to use VML on upcoming projects.

D. Phoenix

With the cancellation of the Mars 2001 Lander after the loss of MCO and MPL, one of the VML 1.0 missions failed to fly. Fortunately, the mission hardware and software were resurrected as the Phoenix mission after years in storage. Due to the operational complexity of the surface phase and experience with VML 2.0 on MRO by LM personnel, the mission chose to replace the VML 1.0 code in its inherited software baseline with VML 2.0. This upgrade addressed a number of concerns, including the incremental loading of very large block libraries to avoid starving lower priority tasks, the decreased memory footprint of instructions, the ability to size engines with different storage space, extra built-in functions, and the repair of identified software flaws.

Phoenix features 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 is state-driven. A series of 24 blocks, as a group, compose the mainline set of EDL activities. The mainline blocks (or segments) make 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.

In parallel with the mainline sequence, all non-critical EDL activities execute on five other engines. These activities include communications, instrument management, CPU monitoring, uplink loss protection, and battery charge optimization. The mainline segments set sequencing global variable values during descent in order to signal to the secondary blocks and coordinate activities. This approach was so successful during testing that the EDL mainline segments have remained unchanged since before launch. The reduction in risk of using the same mainline
segments unmodified for a year before landing has been tangible, as all testing done in this period has the same known timing and subset of configuration.

In keeping with the close coupling between flight and ground that was part of VML's origins, VML's creator and principle author of this paper served as the sequence developer for EDL. This placed both the author and the mission in a unique position, allowing a more in-depth understanding of its operations use. The lessons learned in EDL sequencing are being applied to VML 2.1.

VI. VML 2.1: Anticipating needs

A. New Features

VML 2.1 features major incremental upgrades relative to VML 2.0. Like other VML versions, these changes are based on inputs from VML 2.0 missions like Phoenix and MRO, the author's own experiences developing blocks for Phoenix and other missions, and the need to address new instrument-oriented projects. The feature set in VML 2.1 includes full upward compatibility with VML 2.0, to the point where blocks and sequences developed for VML 2.0 missions load into VMLFC 2.1 and behave identically. Like other VML versions, VML 2.1 also enables missions to come up to speed rapidly by baselining previous mission capabilities, then extending them.

B. State Machines

The success of the state-oriented approach to sequencing complex, mission-critical activities like EDL on Phoenix has led to the inclusion of state machines in the syntax of VML 2.1. State machines consist of named states that transition to other states when conditions become true. A partial example appears in Figure 8. State machines introduce a local scope for attributes, states, and transitions fully internal to themselves.

A well-defined transition called enter exists in order to specify the starting set of activities for the state machine. This also allows states of state machines to themselves be state machines, without any sort of entry and exit point complexity.

State machines contain attributes, identical to the module-level variables featured in previous versions of VML. These attributes provide persistent storage during the execution of the state machine.

State transitions may be performed via \texttt{JUMP\_TO} or \texttt{TAKE} statements. \texttt{JUMP\_TO} is used when the transition from one state to another is simple and does not have side effects as spacecraft commands may, and does not need to coordinate across state machines. \texttt{TAKE} is used when a named transition with the given type is available. This named transition may share synchronization signaling conditions with other state machines, and will only be taken when all state machines are ready. All state machines then simultaneously follow the transition to end up in new states.

\begin{verbatim}
STATE_MACHINE flight_director
ATTRIBUTES
  DECLARE COLLECTION manager_list := {#orbit_det_manager, #imaging_manager, #att_est_manager, ...}
  DECLARE LOGICAL maneuvering := FALSE
END_ATTRIBUTES
...
TRANSITION enter
  BODY
  JUMP\_TO quiescent
END_BODY

STATE quiescent
  BODY
  SELECT\_LOOP
    WHEN gv_fd_next_state = #launch JUMP\_TO launch
    ...
    WHEN gv_fd_next_state = #orbiting JUMP\_TO orbiting
  END_SELECT\_LOOP
END_BODY

STATE orbiting
  BODY
  CALL enable_managers {#orbit_det_manager, #imaging_manager}
  SELECT\_LOOP
    WHEN gv_fd_next_state = #quiescent JUMP\_TO quiescent
    WHEN gv_fd_next_state = #safe TAKE safe_spacecraft
    WHEN maneuvering && gv_fd_next_state = #otm JUMP\_TO otm
  END_SELECT\_LOOP
END_BODY

end_state_machine
\end{verbatim}

Figure 8. State machines. These sample states for a notional mission contain commands and other side effects, and transition to other states when programmed conditions come to pass.
JUMP_TO and TAKE implementation is very similar to removing execution of the current state from the engine and spawning the new state or transition to that same engine. SELECT_LOOP is used to detect conditions and take actions in response to those conditions. In a state machine, the SELECT_LOOP enforces logical conditions as being true before jumping to other states or taking transitions.

The implementation of VML state machines mirrors specifications available in UML state machines. Because of their discrete states, state machines can be analyzed for correct behavior much more easily than can procedural languages. Rather than act as a model to be implemented on top of a procedural language, VML state machines are able to execute directly on a virtual machine, removing any possibility of miscoding the state transition behavior. Doing so also reduces development time and constrains the potential set of actions down to a manageable, easily analyzed number.

It should be emphasized that the standard procedural approach utilizing blocks and sequences is still available, and can be intermixed with state machine use. Such an approach is better suited to a wide number of problem domains than are state machines alone, and is in keeping with VML's "evolution, not revolution" strategy.

C. Objects

Like state machines, objects provide a naming scope that allows capabilities to be packaged together. They feature attributes and locally scoped blocks. An example appears in Figure 9. Objects frequently have a thread of execution associated with them, but are not required to have such a thread, and may instead be used as a convenient package for functionality akin to a library.

Object methods are visible globally using a reference that includes dot structure much akin to C structures or Ada records. This allows the methods within different objects to feature the same local names, but be explicitly unique. For example, an object corresponding to running a spectrograph might feature methods for powering the instrument up and down called on and off. From outside the object (e.g. from an absolute sequence) the full name of the methods would be given in the form telecom.on or spectrograph.on in order to differentiate between the two. Potential naming conflicts between globally visible blocks and object method short names are resolved using scope. The method is local to the object, so the method is locally matched first before a wider global search. Therefore, a local method overrides a global block. This mirrors typical object-oriented design found in languages like Smalltalk.

D. Collections

Collections are arrays of heterogeneous data. Rather than requiring all elements to be of the same data type, each element of the collection can be of whatever data type the user desires. This allows collections to serve in two capacities: as an array (e.g. numbers representing a vector), and as a data structure of related fields. Collections containing heterogeneous and homogeneous data types are shown for a notional block of a notional mission in Figure 10.
Like all other VML data types, the literal representing a collection has an unambiguous format. Like a set in discrete mathematics, collections are bounded by pairs of open and close braces {}, permanently specifying the dimensions of the collection even when copies of it are made and passed as parameters. The default value also serves to specify the collection format, since each of the fields is defined by a literal having a known type. There is no need for a separate data type definition statement, a departure from syntax in traditional strongly typed procedural languages such as Ada or C.

Collections may contain collections, making it possible to pass complex data representations as single parameters between functions. Elements of a collection are accessed using open and close brackets [] and an index, as in Ada and C. Collection elements are numbered starting at 0.

Figure 10. Collections. These sample collections within a block for a notional mission contain a variety of different data types. The local variable quaternion is an example of uniform data types, similar to an array. The local variable obs is an example of non-uniform data types, similar to an Ada record or C structure. Note that the format of the collection is implied by its initialization value, and no separate data typing is required. Field dereferencing is numerical.

The calling function observations contains a collection definition q for a four-element collection, all the elements of which are doubles. This local variable receives a copy of the quaternion local variable when the observe block returns. Since collections, like other data types, can coerce to all data types, a mismatch between collection sizes is handled gracefully: extra elements would be ignored, and missing elements would be set to the value UNKNOWN.

E. Reconfigurable On-The-Fly

One of the most vexing problems in any general purpose software system is to make sure that components are sized correctly. Sizing in VML includes the number of engines, the number of instructions on each engine, the number of operands on each engine's stack, the amount of space for storing strings and spacecraft commands on each engine, and so forth. In preceding versions of VML, all of these values have been fixed at compile time of the underlying flight component. VML 0.0, 1.0, and 1.1 flight components all featured identically sized engines. This was problematic if block libraries grew, because the memory footprint of the flight code was magnified by the number of engines, even if the need was specific to one engine. VML 2.0 improved upon this by allowing different memory allocations on different engines, but the operations development teams frequently outgrew the allocations. In order to resolve these problems, VML 2.1 allows the sizing of each engine to be changed at runtime. The overall number of instructions and operands, and the amount of string and command space, are fixed at compile time, but may be reallocated among engines in order to more suitably meet unanticipated needs.

F. Command Interface Enhancements

Controlling the activities of the VML 2.1 Flight Component requires only a few extensions over the existing set of commands for 2.0. All of the commands shown in section III J are supported in VML 2.1. Commands have been
added to allow setting flight software attributes which govern elements like access timeouts and command modes. This feature is also generalizable to flight software variables outside of the VML flight component. In addition, commands have been added to support dynamic reallocation of engine attributes. The new commands are listed below.

- **VM_ATR_SET_DBL**: set an attribute to a given double floating point value
- **VM_ATR_SET_INT**: set an attribute to a given integer value
- **VM_ATR_SET_STR**: set an attribute to a given string value
- **VM_ATR_SET_UINT**: set an attribute to a given unsigned integer value
- **VM_ALLOC_INSTR**: set the instruction allocation count for a given engine
- **VM_ALLOC_STR**: set the string space allocation for a given engine
- **VM_ALLOC_CMD**: set the spacecraft command allocation for a given engine
- **VM_ALLOC_RESET**: remove all preceding allocations received
- **VM_ALLOC_APPLY**: apply all allocations received in preceding commands

### VML 2.1 Missions

**A. AutoGNC**

This technology demonstration program builds on the optical navigation work for the successful Deep Space 1 and Deep Impact missions. The name stands for Automated Guidance, Navigation, and Control, and intends to create prototype hardware to allow any spacecraft to self-navigate to any given body, following an ephemeris model. Target missions include orbiters, landers, asteroid body interceptors, and cometary interceptors. Different modes of operation are supported, including stellar navigation, inertial navigation, and optical navigation.

VML 2.1 serves as the executive of the overall software system. A flight director implemented as a state machine enables and disables managers of various system subcomponents, depending on mission phase. The managers coordinate together using rules and global variables in order to issue commands to instrument components. VML treats the AutoGNC instrument as its own small spacecraft, with commands, uplink, and telemetry. This approach has allowed rapid implementation of very complex control logic.

**B. Juno**

Juno, the Jupiter Polar Orbiter, will be a VML mission. Juno is in the early stages of development and system engineers are performing a trade study to decide whether to use VML 2.0 or advance to VML 2.1. There has also been interest in the possibility of installing VML in one or more of the instruments as an instrument executive. The work continues.

**C. GRAIL?**

GRAIL, the Gravity Recovery and Interior Laboratory lunar mission, has recently been approved to proceed with Phase B. It will be a Lockheed-Martin built pair of spacecraft with non-NASA heritage hardware but NASA-heritage software. JPL will be starting the process of assessing VML inclusion in the project plan in the next few months.

### VII. Arcs of Development

**A. Arcs**

At each step of the way, VML has been carefully crafted to provide features required by missions without imposing one-off implementations. Every VML feature has been created with an eye toward generic capabilities useful on current and potential future missions. Features are not removed. Rather, as the arc of deep space missions has proceeded, VML has been extended where necessary without sacrificing past capabilities.

Up to this point, we have examined the story of VML from the point of view of missions. We will now look at VML from the point of view of arcs of development, presented in Figure 11. VML features may be considered as groupings of data structure elements, language elements, and execution elements. Each of these elements is interesting in its own right, and shall be examined separately. The figure the points in time at which development arcs are considered complete, and correlates these features with the various missions using versions of VML.
Data are elements that include data types, name scopes, externalized data access, spacecraft command parameter types, and structuring of data. The language arc includes the VML scripting directives used by operators to express executable concepts, including function layout, timing, programming constructs, and expressions. The execution arc encompasses the hidden implementation details that allow the data and language arcs to be implemented as real software running on real platforms, including processor requirements, sizing, memory alignment, and support software needs.

### B. Data Arc

Over the past decade, the data types supported by VML have grown in number and complexity. Variable scopes and the ability to access data from the overall flight software build have been added. Constant parameters for spacecraft commands and support for CCSDS header fields have been folded into currently flying missions. The concepts of objects, object-level scoping, data structures, and array elements have also been introduced.

The basic VML 0 sequencing capability featured only the data constructs necessary to replicate the intended functionality of the original Stardust, MCO, and MPL requirements. Data was expressed as globally accessible integers, plus a small per-engine set of locally accessible integers. Variables had no names, but were instead identified using indices within the simple pre-VML sequence specification. Integers represented absolute times to a resolution of seconds, or relative times to a resolution of tenths of seconds. Integers also performed as logical values, with 0 indicating false and non-zero indicating true. No floating point or string representations were available.

Under VML 1, a wide variety of different data types was incorporated, allowing integers, unsigned integers, Boolean logicals, double precision floating point values, strings, and time. In addition a special value "unknown" was available to represent missing parameters. Time was represented by two 32 bit unsigned integers taken together to represent mission time in ticks, allowing absolute and relative times to be known to the same resolution, and therefore to be easily compared and calculated. All conversions between variable types were legal and defined, including conversions from numbers to strings and vice versa, making polymorphism during function calls simple and safe. Default values were required as part of all variable declarations, removing the potential for errors due to uninitialized variables. Variable scopes were enhanced with local, module level, and global variables. The module level variable scope allowed variables to hold values between invocations of functions in which they were assigned. This capability was useful in its own right, but would form the basis for objects later on.

A minor enhancement to VML was made to address Spitzer uplink concerns. Incorporating an optional table of strings and a specialized compressed time format for small time values saves roughly 50% of the uplink bandwidth for the mission by reducing the size of blocks and sequences. This allowed Spitzer to live within its uplink allocation while simultaneously maintaining compatibility with the original in-line string format used in test products.

Figure 11. Arcs of development. The basic elements of VML data, language, and execution features are listed in order of development, along with a list of mission time spans showing use of the various versions.
Experience on the Odyssey and Spitzer missions led to further data enhancements under VML 2.0. Managing global variables on these missions consisted of creating a set of named global variables with specific data types for known uses, then adding spare variables whose use was to be determined during flight. All sequence global variable definitions had to be complete before compiling and linking the final spacecraft flight software load. Because the name of the spare did not reflect its eventual use, some confusion arose about the meaning of spare global variables that were decided on well into the mission. Therefore, a VM command was added to rename global variables during runtime, allowing the names of these spare variables to be changed in flight. This feature has been exploited on MRO and especially on Phoenix, as Phoenix surface operations blocks were designed and implemented primarily after completion of the final flight software build.

Much of the need for sequencing global variables is driven by the need for blocks to respond to flight software conditions. However, it is difficult to anticipate all such interactions ahead of time. Therefore, MRO operators requested the ability to map telemetry data into VML global variables. Since knowledge of spacecraft state is ultimately pushed into telemetry, direct access to that information allowed blocks to respond to any element of spacecraft state without modifying the flight software to push that information into global variables. This external data access capability allows any spacecraft flight software data to be accessed in a generic fashion.

Spacecraft operations on both Dawn and MRO required the ability to dynamically build and dispatch commands featuring unchanging parameters. In Dawn's case, these parameter values represented fields of CCSDS packets, whereas for MRO, the fixed constant parameters were needed for instrument commands. The needs in both cases were identified within five days of each other, and in keeping with VML's generic implementation approach, the same underlying implementation applied to both.

The modern concept of objects with attributes made its way into VML 2.1, built on module variables and a simple nested naming convention similar to record access in C and Ada. Similarly, states for state machines built around UML state transition diagrams use the object baseline. Collections, the single most-as-asked-for feature, are aggregations of data similar to data structures or arrays, created using object-like aggregation techniques to maintain cohesion.

C. Language Arc

The VML 0 language facilities included a programmable delay, branching on true / false conditions to labels expressed as integers, branching unconditionally to labels, and evaluating simple unary and binary operators. Spacecraft commands were isolated from the language definition in order to reuse code between missions. The existing ground software produced binary sequence loads, so a translator was incorporated into the flight software to dynamically translate these binaries into a runtime load acceptable to the VML flight parser. No compiler was necessary for VML 0.

With the deployment of VML 1.0 on Odyssey, Spitzer, and Genesis came the full set of initial VML capabilities in both flight and ground. The VML Compiler translated fully implemented human-readable VML script into uplinkable binaries, replacing the onboard binary translation in VML 0. (The translator remained installed in the flight code in order to accommodate Genesis mission design demands for a ground system identical to Stardust's, however.) Arithmetic and comparison operators were extended to operate on the new data types. Built-in functions for string manipulation were added, including sizing, concatenation, and substring extraction. Names for blocks and sequences allowed more than one per file, giving rise to the concept of libraries of reusable on-board blocks that could be invoked repeatedly. WHILE loops and IF statements allowed structured programming and testing of conditions using simple expressions with one comparison operator.

VML 2.0 built up capabilities with FOR loop constructs and compound expressions. Spitzer's need for trigonometric functions had been met somewhat clumsily with finite series calculation blocks, so built-in functions for trigonometry were added to VML 2.0 for simplicity and convenience. Protection was added against spawning blocks and sequences more than once, unless explicitly allowed using a REENTRANT flag. The binary translator for backwards-compatible support of the obsolete Stardust and Genesis ground system was removed with concurrence of the VML 2.0 missions.

VML 2.1 continues the enhancement. Intrinsic time tags make VML look more like mainstream languages, allowing time between statements to be assumed rather than explicitly stated. The implementation of entry, descent,
and landing blocks on Phoenix as cooperating state machines led to the implementation of direct language support for UML-like state machines. Waiting on more than one event via compound event detection grew directly from language support for these state machines. So too did the implementation of SELECT_LOOPs, allowing a very dense representation of possible code branches.

D. Execution Arc

The VML 0 execution engine core supported spawning sequences (either absolutely or relatively timed) to run in parallel on identically sized engines, but no facility was made for passing parameters. Instead, global variables held any values passed between sequences, requiring operators to exercise care to prevent collisions in variable use when spawning multiple parallel sequences. The engine core was compatible only with VxWorks running on PowerPC-derived processors with big-endian byte ordering. A parsing buffer was incorporated to allow large sequence loads to be incrementally parsed from memory rather than directly from the file system, increasing the speed of loading.

VML 1.0 introduced the use of named functions to represent sequences and reusable blocks, enabling more than one block to be present in a file loaded onto an engine. Hand in hand with named functions came the CALL operation with parameters, whereby information could be passed into and out of reusable blocks in order to affect block behavior. The CALL functionality eliminated the need for complicated user-written synchronization mechanisms using global variables, implicitly causing the calling function to remain suspended until the called sequence completed. Block library routines were used extensively by Odyssey for aerobraking and mapping. Spitzer also made considerable use of block libraries for performing observations and commanding communications passes. Dynamic commands were used in conjunction with blocks in order to allow the spacecraft to substitute calculated variable and parameter values into spacecraft commands, coupling the VML sequencing system directly to command dispatch and dramatically simplifying block development.

Event detection allowed VML 1.0 to efficiently recognize spacecraft conditions represented by the flight software, and allowed non-deterministic timing in blocks. This facility found use aboard Spitzer to allow observations to proceed as quickly as possible after completing slews and settling the spacecraft. Odyssey used event detection during the completion of aerobraking as well. The event detection facility would act as the basis for compound event detection and command completion in VML 2.1.

VML 2.0 added a number of enhancements to simplify the deployment of VML on non-Lockheed missions by using an interface layer and an abstraction of real-time operating system calls. Four-byte alignment made the underlying data structures compatible with Dawn's flight computer, and allowed the flight code to run within the OLVM program on Sun workstations without the use of non-standard C compiler flags. Compile time sizing allowed engines to be specifically tuned for uses, letting some engines be oversized in order to hold block libraries and others to be sized for simple instrument operations: MRO, Dawn, and Phoenix all made extensive use of tuned sizes. Based on operations experience, protection kept important elements like block libraries from being accidentally unloaded. Reservation allowed engines used in specifically-targeted mode to be separated from engines used in first-available mode, reducing the number of engines needed by allowing certain activities to be allocated from an available engine pool. Incremental parsing reduced the allowed CPU time used for loading to be spread out, reducing the load on the system, removing the possibility of starving lower priority tasks during file loads, and allowing larger files with more blocks to be loaded.

With the advent of VML 2.1, the concept of command completion was borrowed from the Mars Exploration Rovers, allowing function timing to be inherently coupled with command execution timing and simplifying block implementation. In response to operations requests, runtime resizing of engines and commands allowing the direct setting of attributes has been added. Finally, to increase the number of compatible target platforms, the newest code base supports Intel-type little-endian byte ordering.

These arcs of development do not exist separately from one another. They intertwine and support each other, giving rise to the overall set of reusable capabilities that is VML.
VIII. Conclusions

Virtual Machine Language is a powerful tool. Through its eleven-year life to date, it has provided a modern, flexible, scalable software set, enhancing mission success over a large range of missions. The simple abstraction of complex spacecraft operations into parameterized, reusable blocks streamlines operations and reduces uplink volume and risk while enhancing science efficiency. The configurable sequencing capability allows missions to customize the software suite to their needs while operating within the multimission environment and experience base. The high level capabilities of VML and the tight integration of the flight and ground components speed development time while fostering "test-as-you-fly" conditions. The ability to read and act on prechannelized telemetry opens the door to many kinds of onboard automation.

Increasingly more capable versions of VML are flown with each new set of missions. Lessons learned come from testers and operators with real flight experience, leading to the incorporation of modifications and enhancements based on those lessons. Missions use a preceding mission's flight software and configuration as a starting point, significantly reducing the amount of time required to produce the first functional flight software build. Operations builds upon blocks created for preceding missions, making modifications as appropriate, and applying the same sequencing architecture. For sequence software, at least, the era of parallel yet separate software development is over. With the coming capabilities of state machines, new platform compatibility, and instrument based executives, it is clear that VML's arc of development will continue on for quite some time to come.

Acknowledgments

The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under an agreement with the National Aeronautics and Space Administration.

References

Reports, Theses, and Individual Papers


Related web sites

Blue Sun Enterprises VML Website http://www.bluesunenterprises.com/vml
Mars Reconnaissance Orbiter Website http://mars.jpl.nasa.gov/mro/
Dawn Mission Website http://dawn.jpl.nasa.gov/
Juno Mission Website http://juno.wisc.edu/