The Fully Programmable Spacecraft: Procedural Sequencing for JPL Deep Space Missions Using VML (Virtual Machine Language)[1][2]

Dr. Christopher A. Grasso
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive
M/S: 301-250D
Pasadena, CA 91109-8099
303-641-5926
cgrasso@stellarsolutions.com

Abstract—VML (Virtual Machine Language) is an advanced procedural sequencing language which simplifies spacecraft operations, minimizes uplink product size, and allows autonomous operations aboard a mission without the development of autonomous flight software. The language is a mission-independent, high level, human readable script. It features a rich set of data types (including integers, doubles, and strings), named functions, parameters, IF and WHILE control structures, polymorphism, and on-the-fly creation of spacecraft commands from calculated values.

VML has been used on six JPL deep space missions. It is currently in use on Mars Odyssey, Stardust, Genesis, and the Space Infrared Telescope Facility (SIRTF). It is slated for use on the 2005 Mars Reconnaissance Orbiter. The language and associated flight code has allowed spacecraft operations teams to place autonomy aboard deep space missions, implemented as operations products (blocks and sequences).

The flight component of VML is implemented in C. This flight code interprets binaries produced by the ground component. It runs on a scheduled basis (typically 10 Hz) within the flight code, providing a configurable number of parallel threads of execution (typically eight, up to 65,533). The flight component responds to commands to load files, begin and end execution, save and restore global variable values, and set error responses. The VML flight component can issue spacecraft commands using parameter values and local variables within blocks and sequences. Hooks can invoke special-purpose mission-specific flight code from sequences.

This paper lays out language constructs and capabilities, code features, and VML operations development concepts. The ability to migrate to the spacecraft functionality which is more traditionally implemented on the ground is examined. The implications for implementing spacecraft autonomy without the need for expensive flight software agent development is also discussed.

TABLE OF CONTENTS

1. INTRODUCTION: SEQUENCING

Spacecraft frequently require some ability to perform actions initiated by commands. Commands may be immediate (directly executed after being issued by ground control) or timed (executed at a specified time after uplink to the spacecraft). The execution of timed spacecraft commands is known as *sequencing* [1].

Sequences are typically represented in a planning interface within the ground system, translated to an uplinkable form, radiated to the spacecraft, loaded by some means, then executed. Sequence execution results in the issuance of commands to the spacecraft in some timed order.

Some older spacecraft (for instance, the Solar Mesosphere Explorer [2] [3], flown by the University of Colorado) used hardware sequencers, in which the flight computer managed time tags directly attached to each command, and checked for command times using logic circuitry. More modern spacecraft with real-time operating systems and preemptive task scheduling implement sequencing as a software component. This paper focuses on this latter arena.

The features implemented in the generic flight software sequencing capabilities profoundly affect the complexity of operating the spacecraft, the size of the team necessary implement sequences, the operations able to be undertaken, the frequency of uplink, and the size of uplinked products.

[1] 0-7803-7231-X/01/$10.00/©2002 IEEE
[2] IEEEAC paper #187, Updated September 28, 2001

Virtual Machine Language (VML) sequencing carries a number of distinct advantages over more traditional sequencing architectures, in both capability, personnel time, and cost. This paper will describe VML components, capabilities of the language, its use on NASA JPL deep space missions, and implications for spacecraft autonomy.

2. TRADITIONAL SEQUENCING MODEL VS. VML MODEL

In general, spacecraft sequencing can follow one of two models: timed command or procedural.

Timed command sequencing

The more traditional (non-VML) timed command approach uses a fixed set of timed command locations which are continually checked to see if any particular command is due at any particular time, illustrated in Figure 2-1.

Time tag	Command
Time tag	Command
Time tag	Command
...	
Time tag	Command
Time tag	Command

Instruction storage

Figure 2-1: Traditional sequencing instruction store

All due commands are identified and fired off on the same clock tick. This approach does not segregate commands to be issued into functional units (i.e. routines), and the parallel nature of the execution makes developing parameterization, logic, and decision-making problematic. Branching to labels is sometimes implemented in this model, but the resulting sequences can feature spaghetti logic and unintended code path interactions. In addition, the lack of functional units results in a lack of private variable space. All variable use for calculation must be allocated and checked for collisions between desired activities. Enforcing separate spaces for all but the most rudimentary activities becomes difficult, and is not inherent to the model.

Procedural sequencing

By contrast, the sequences developed in VML are procedural in nature. At any particular time, only one instruction is considered to be "next" on a sequence engine. This allows named sequences which can be called using parameters, easy creation and evaluation of logic constructs, and an implicit ability to branch and loop. Parallelism is achieved by instantiating a fixed number of sequence engines, and explicitly loading and running sequences as threads on those engines. These kind of sequence engines are called virtual machines. They resemble a CPU which can interpret instructions, with memory, dynamic data storage implemented as a stack, and an instruction pointer (see Figure 2-2). Some number of machines are instantiated for the mission. These machines limit the number of threads of execution which can operate in parallel.

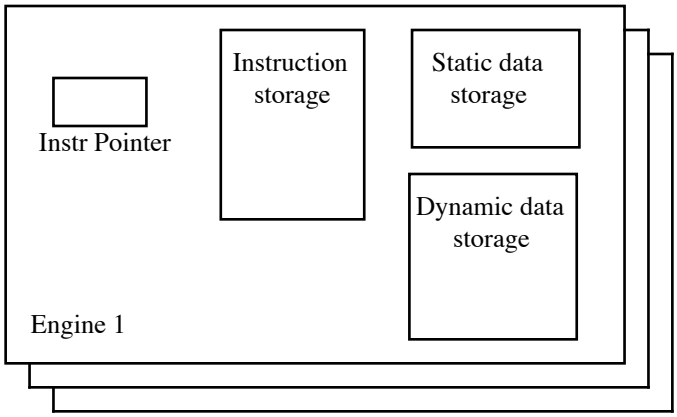

Figure 2-2: Virtual machine sequencing instruction

Each engine is used for two distinct purposes: storing sequences, and executing sequences. When a file containing a VML module is loaded into an engine, the named sequences (called *functions*) within that module become available for running on any engine. They are invoked by name rather than in index. In some cases, the function is executed on the same engine in which it is stored, as shown in Figure 2-3.

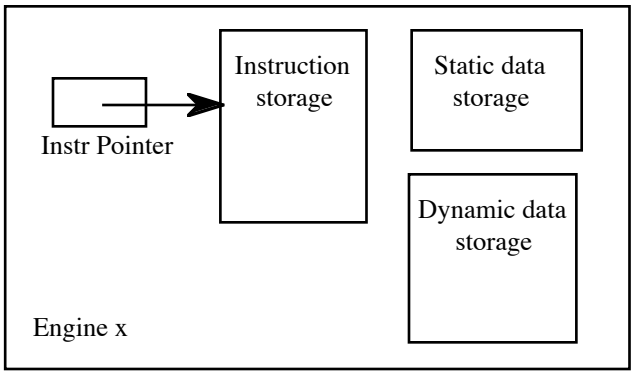

Figure 2-3: Function running on same engine as stored

In other cases, the function is executed on a different engine than the one in which it is stored, as shown in Figure 2.4. This would be the case for a function calling another function in a library stored on a different engine.

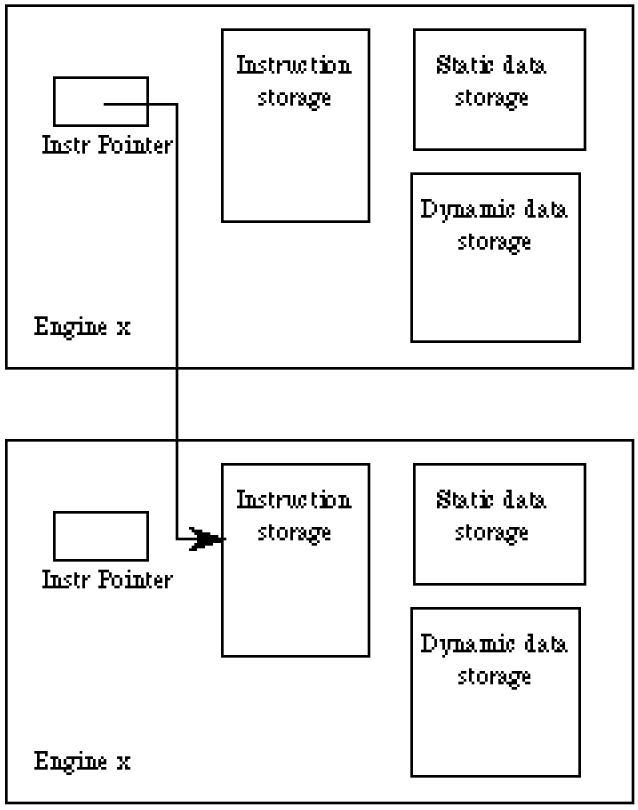

Figure 2-4: Function running on different engine than stored

3. VML COMPONENTS

The Virtual Machine Language system consists of several components working in concert to provide a programmable spacecraft capability (see Figure 3-1 below).

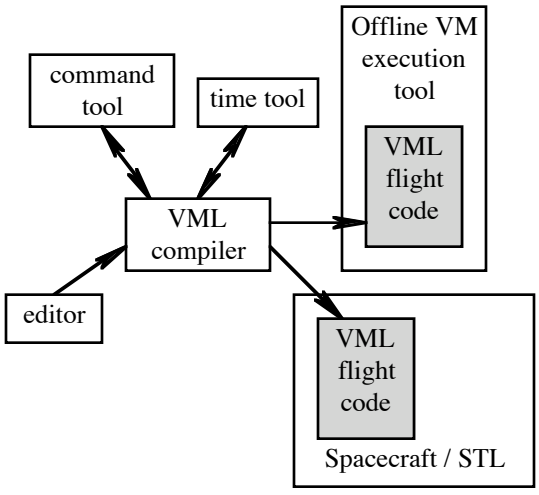

Figure 3-1: VML components

VML compiler

The user creates functions as text using a text editor, or a front-end generating tool. The text is translated by the VML compiler according to generic VML constructs and mission-specific definition files for allowed global variable names and constants.

The VML compiler also uses mission-specific command translation and time translation tools. The command tool takes a human-readable string representing a command to be translated, and passes back the binary value able to be interpreted by the spacecraft command software. The time translation tool is invoked whenever an absolute time is encountered in the sequence. The time translation tool reconciles spacecraft and earth time in order to account for clock skew in the sequence products.

The VML compiler is a program which runs within the ground system on a Solaris 7 platform. It has been developed as JPL Category A SEI level 3 code, with appropriate methodology, documentation, review, and testing.

VML flight component

The resulting binary is ingested by the VML flight component. The flight component is used in two environments: as flight software running in a test lab or on the actual spacecraft, and in a faster-than-real-time execution environment called Offline VM (below).

The VML flight component runs as an embedded task under VxWorks or similar real-time operating system within the flight software. It works in concert with the rest of flight software, dispatching commands to other flight software tasks in order to affect changes to the spacecraft behavior. The flight component has been developed in a manner compatible with JPL Category A SEI level 3 code, with appropriate methodology, documentation, review, and testing.

Offline VM

Offline VM is a faster-than-real-time tool which runs in the Solaris 7 Unix environment. Originally developed as an informal flight software testing aid, it has proved sufficiently useful as a sequence test tool that it is currently being redeveloped as JPL Category A SEI level 3 code, with appropriate methodology, documentation, review, and testing.

Offline VM encapsulates the flight code with a user interface. It is able to ingest sequences that have been produced by the VML compiler. The clock within Offline VM is harnessed and entirely under the user's control. Virtual time in this environment can progress several thousand times real-time. This allows the user to progress time for a known period, stop and examine the state of the sequence engines, perform disassemblies of sequence

binaries, study command issuance times, inject spacecraft behavior into the state of global variables, and resume execution at will.

Since Offline VM runs on an ordinary workstation, it is an easily distributed and very low-cost tool. The high expense of maintaining and coordinating access to a full-up software test lab (STL) featuring a real flight computer makes Offline VM an attractive alternative for iteratively developing and debugging sequence logic. Mars Odyssey and SIRTF use Offline VM in exactly that fashion, performing dozens of development iterations in the Offline VM environment before testing the sequence in the expensive and over-subscribed STL.

4. VML CONSTRUCTS

The procedural orientation of virtual machines allows sequences to be expressed using a number of high-level language constructs. These constructs form a simple but powerful scripting language in which users can express desired spacecraft activities using named functions, with parameters, a variety of data types, and a rich set of operators.

Modules

A module is a container for one or more functions, along with optional persistent storage.

Modules partition the functionality of the problem space into manageable chunks. For instance, a library module may contain several reusable blocks for controlling instruments and performing communications activities. A daily absolute sequence may reside as the only function in another module, since it is changed out with regularity.

One module exists per file input to the VML compiler. This same module is defined in a flight-compatible format in the VML compiler output file.

Functions: Blocks, Relative and Absolute Sequences

Function is the generic term for a sequence containing absolute and/or relative time tags. A function is a named executable chunk of VML which may accept parameters. Functions may also return values to a calling function.

An *absolute sequence* contains one or more statements with an absolute time tag. Absolute sequences may be run only once: since time always increases, executions after the first execution will cause instructions to be late.

A *block* is a function that is intended to be reused (and frequently is stored onboard the spacecraft). Blocks contain only relative time tags so that when they are executed none of the statements are late.

A function that contains only relative time tags between its statements and is not intended for multiple uses is called a *relative sequence*. Relative sequences contain only relative time tags so that they may be kicked off at any time without being late. Relative sequences are frequently autogenerated by ground-based tools for activities like aerobraking maneuvers or daily operations which are planning dependent.

Time

Time may be specified in several formats: as absolute (wall-clock time), spacecraft time in seconds, and relative time in hour/minute/second form. Each instruction has a time tag which acts as a delay between its execution and the completion of the previous instruction.

The flight software component of VML executes statements at a time resolution configured at compile time, and frequently set to 0.1 seconds. Zero time tags (no delay) are allowed between instructions: they will execute in order, but on the same time tick.

In addition, mathematical operations on time are available for calculating delay values needed in DELAY_BY and DELAY_UNTIL statements.

Variables

Several different scopes of variables exist in VML sequencing: local, module, and global.

Local variables are defined within functions, and are not visible by name outside the function. Each instance of an executing function contains fresh copies of its local variables.

Module level variables are visible by name to all functions defined in the module, and have persistent data values until the module is unloaded from the engine.

Global level variables are visible by name to all functions, and to flight software. Storage in global variables is persistent. Global variables are used for event-driven sequencing, allowing sequences to respond to environmental changes in the spacecraft.

Variable types include integer, unsigned integer, floating point double, logical, time, and string. Variables may be assigned regardless of type: all assignments of different types result in meaningful values to the assignee. This runtime flexibility removes many constraints with which operators would otherwise have to deal, and results in less complicated sequences.

Operators

A wide variety of operators is available in VML. Arithmetic operators include absolute value, negation, addition, subtraction, multiplication, division, modulo, and power. Bitwise operators include and, or, exclusive or, invert, shift left, and shift right. Logical operators include and, or, exclusive or, and not. String operators include length, split left (returns substring from start of string up to and including given character position), and split right (trailing substring starting at given character position).

At present, the most complex expression is one including a binary operator, e.g.

```
R00:00:00.1     max := 15.5 + try
```

Expressions with precedence may be incorporated on a future mission.

Conditionals

VML includes an IF construct which can be used for choosing a code path based on variable values. This selection allows logical evaluation of multiple conditions using ELSE_IF and ELSE statements. The IF construct is particularly useful for reacting to parameter values passed into a function, calculated local variables, and global variable values.

Loops

A WHILE loop is available for structured conditional looping. This construct can be used to repeat a set of statements until a condition becomes TRUE or FALSE. This construct also allows repeating a set of statements a specific number of times using a counting variable.

Event-driven Sequencing

The WAIT and TEST_AND_SET statements are used to detect events represented by global variables.

The WAIT statement suspends operation of the function until its condition is met, then resumes execution of the function at the next instruction. This instruction is particularly useful for non-deterministic sequences which are related to real-time events: rather than assuming worst-case timing, the sequence can be designed to execute with a minimum of wasted time.

TEST_AND_SET is used on a counting semaphore for managing a shared resource. This is a classical real-time programming access problem [4]. An example use might be to enforce mutually exclusive access to an instrument suite by two separate, non-deterministic sequences. Using a check with a conditional followed by a subtraction leads to an intractable race condition whereby both blocks could complete the IF check before setting the semaphore with the blocking value. TEST_AND_SET allows an integer global variable to be checked and decremented in one instruction, preventing this race.

Both event-driven constructs allow an optional timeout to prevent infinite waits and force an upper bound on execution times.

Call: in-line function execution

A function may be executed in-line from another function using the CALL statement. The calling function is suspended, the caller is executed, and then the calling function resumes. The caller may pass parameters to and receive return values from the called function as appropriate using a RETURN statement. Relative time tags for the statement after the CALL indicate the amount of time from the completion of the CALL statement.

Calling does not start a separate thread of execution or use another sequence engine. Instead, resources on the *calling* engine continue to be used to maintain the thread of execution. Refer back to figure 2-4. This figure shows an engine using code that is stored on another engine (e.g. a master sequence calling a block in a library). The instruction pointer contains a value that indicates code residing on a different engine, but the instruction pointer itself resides on the same engine.

Calls may be nested arbitrarily deeply, limited only by data stack space on the calling engine. However, call depth greater than about three become difficult to evaluate, and can make understanding the timing of the sequences problematic.

Spawn: New Thread of Execution

A new thread may be created to run in parallel with existing threads using the SPAWN statement. The spawning function may pass parameters to the spawned function, but no return value is possible. Unlike CALL statements, SPAWN statements complete on the same tick of the clock at which they are invoked. The spawned function is scheduled for evaluation on the next time tick.

Spawning is useful when an activity needs to be initiated that is functionally separate from the initiator, and contains no intrinsic ordering requirements relative to the initiator. For instance, a master sequence may need to initiate downlink at a certain time, but continue to manage instrument observations. If the downlink activities are consolidated in a block, the master sequence can simply spawn the downlink block, then continue on with its usual management tasks. This approach simplifies development of the master sequence by eliminating the interleaving of activities within the body of the sequence. It also allows the functionality of the downlink activity to be abstracted into a block, tested, then repeatedly used.

Constant Commands

Commands are specified either in a untranslated form, which can be passed to a mission-specific translation tool, or a pretranslated form, wherein some tool has specified the binary pattern of the command for inclusion in the sequence. Command syntax varies from mission to mission, but VML can handle any commanding regime so long as the binary size of the command can be calculated.

Dynamic Commands

Commands may be built on the fly by the VML flight component based on parameter and variable values. Any command defined in the system that can be interpreted by the flight software can be built with a special external call "issue_cmd". Dynamically built commands values are validated according to the same rules built into the ground system, thereby protecting the spacecraft from miscalculations. Invalid command parameter values result in a command error and will abort a thread of execution if aborts are enabled.

5.0 SAMPLE CODE

Below appears a module containing a sample block called acquire_star that is part of a larger library. Other blocks in this module have been abstracted away by the ellipses (…). This example shows most of the major constructs of VML, including parameters, local variables, a variety of variable types, comments, loops, conditionals, dynamically built commands, calling, and event-driven sequencing. Each statement in the body of the block has a time tag expressed in time format down to tenths of a second, with timing relative to the completion of the previous statement.

```
MODULE

BLOCK acquire_star
    INPUT ra    ;right ascencion
    INPUT dec   ;declination
    INPUT file
    INPUT acq_failure_delay
    DECLARE INT try := 0
    DECLARE INT acq := -1
    DECLARE DOUBLE max := 0.0
BODY
R00:00:00.1 WHILE try < 3 DO
R00:00:00.4     ISSUE STAR_TRACKER RESET, 0x32
R00:00:05.4     EXTERNAL_CALL "issue_cmd" "ACQ",ra,dec,file

R00:00:00.1     max := 15.5 + try
R00:00:00.1     acq := -1
R00:00:00.0     acq := WAIT gv_star_acq >= 0 TIMEOUT max

R00:00:00.1     IF acq = -1 THEN
R00:00:00.1         CALL record_failure tries, ra, dec
R00:00:00.0         DELAY_BY acq_failure_delay
R00:00:00.1     ELSE_IF acq = 0 THEN
R00:00:00.0         DELAY_BY 11.5
R00:00:00.0     ELSE
R00:00:00.1         return TRUE   ;succeeded
R00:00:00.0     END_IF

R00:00:00.1     try := try + 1

R00:00:00.0 END_LOOP

R00:00:00.1 RETURN FALSE    ;failed

END_BODY

…

END_MODULE
```

The acquire_start block receives four INPUT parameters and has three local variables. It attempts to acquire the star three times, governed by a loop with a counter called try. It issues a reset command to the tracker, then dynamically builds an acquisition command using values passed in as parameters. It calculates a timeout to use, then waits on a global variable which indicates that the star has been acquired. If the acquisition attempt times out, acq remains at its original value of -1, and the failure is recorded by calling another block record_failure. If a value of 0 allows the block beyond the WAIT statement within the timeout period, another attempt is made. If a positive value allows the block beyond the WAIT statement, the block returns with a value of TRUE to its caller, indicating success.

6. SIRTF UPLINK REDUCTION

The combination of parameterization, variables, a large set of data types, and dynamic commanding makes it possible in some cases to reduce the size of uplinked products over traditional ground-expansion of sequences.

In the case of the Space Infrared Telescope Facility (SIRTF), the design of the instruments required that large commands (hundreds of bytes) be transmitted frequently over a serial line. The exact byte patterns would have to be embedded repeatedly in a controlling sequence performing observations, leading to very large ground-expanded blocks which exceeded available uplink contact time through the Deep Space Network.

During most of the commanding, however, most of the parameters in the instrument commands stayed the same. So, rather than ground expand the instrument commands, blocks were created that set the observatory to particular instrument modes, represented by global variable values. Other blocks were created that accept parameters for the instrument command values that varied, and dynamic commands were built from these parameters and the modal global variables. The controlling sequence then invoked a block with one parameter, initiating a cascade of activity which resulted in sending commands to the instruments.

The resulting reduction in uplink allowed the spacecraft to meet its target DSN uplink allocation, without developing complex instrument flight software additions or new sequencing flight software requirements.

7. SIRTF DATA RETURN INCREASE

In some circumstances, nondeterminism in a physical process can cause sequences to be designed with an overly conservative timing, leaving a spacecraft idle when it could be producing data. If there is a lifespan limit to the spacecraft, conservative timing leads to less data from the mission.

The Space Infrared Telescope Facility features both a limited lifespan and nondeterministic processes. SIRTF uses coolant to keep its instruments at operating temperatures. This coolant has a boil-off characteristic that limits the lifespan of the mission. The pointing process for the telescope has some event-driven physical characteristics which make its behavior when settling from target slewing nondeterministic.

Rather than incorporate worst-case timing, SIRTF uses the VML WAIT instruction, watching a global variable which indicates that the spacecraft has settled before proceeding with instrument observations. Estimates place the increase in data collection over the life of the mission as high as 10% due to the elimination of small idle timespans.

8. MARS ODYSSEY AEROBRAKING

Unanticipated events can require a response by the spacecraft in order to maintain safe operations, or even to survive. The lightspeed delays communicating with distant spacecraft can exacerbate the effect of unanticipated threats to spacecraft safety, as the ground is seeing a snapshot of state tens of minutes in the past. Due to its flexible logic and ability to monitor spacecraft state, VML is suited to respond to threatening events

Mars Odyssey performs several months of *aerobraking*, using the atmosphere of Mars to change the orbit of the spacecraft without using large amounts of propellant. During the final days, unexpected atmospheric plumes can dramatically increase the density of the atmosphere, causing the drag pass drop the orbit of the spacecraft by amounts which could lead to an unintended reentry.

Sequences monitor each drag pass and can take action upon an encounter with this sort of drag situation. The response is to autonomously initiate a pop-up maneuver without the need for ground intervention, and before the ground could even physically become aware of the situation. The logic for responding to an atmospheric plume is built directly into the sequences, without requiring expensive development of complex flight software.

9. Implications for Migrating Autonomy

A full-featured, robust sequencing language like VML has some profound implications for the location of decision making in a spacecraft system. The balance between portions of the system making decisions and taking actions (flight software, ground operations, and sequences) can change with the extra capabilities provided by VML.

Because sequences can be developed faster and more cheaply than flight software, and can be placed onboard more easily, small amounts of autonomy developed by the spacecraft operations team can supplement basic flight software capabilities. An example is the aerobraking pop-up maneuver on Mars Odyssey discussed previously.

The ability to make logical decisions based on spacecraft state, calculate values, then take action using dynamically built commands could find utility in fault protection detection and response, autonomous guidance, instrument allocation, and contact management. Using the Offline VM sequence execution tool, operations could start work on developing these kinds of sequences earlier in the spacecraft implementation cycle, well before feature-complete flight software is available.

10. Conclusions

The advanced procedural capabilities in VML simplify spacecraft operations by allowing functionality of the spacecraft to be abstracted into named blocks accepting parameters. Uplink product size is minimized by the ability to call blocks that implement most of the command steps. This block approach also allows some autonomous operations aboard a mission without the development of autonomous flight software.

Procedural orientation allows sequencing to be approached as a structured programming problem, which in turn allows higher quality products to be produced by smaller operations teams. The use of rapid check-out tools like Offline VM reduces the modification cycle time of sequences, allowing the operations development team to produce products on an accelerated schedule. The potential for cost savings in deep space missions using the VML approach is considerable.

Future work includes expansion of the complexity of expressions in VML beyond binary operations, reduction in memory requirements for the flight component, and application of the VML flight and ground components to new missions.

References

[1] D. Kirkpatrick, "Spacecraft Subsystems: Telemetry, Tracking, and Command", *Space Mission Analysis and Design, 3rd Edition*, pp 381-394, edited by J. R. Wertz and W. J. Larson, New York, Microcosm Press and Kluwer Academic Publishers, 2000.

[2] Barth et. al.,*Solar Mesosphere Explorer (SME) Final Mission Operations Report,* Laboratory for Atmospheric and Space Physics, University of Colorado, November 1989.

[3] *Solar Mesosphere Explorer Mission Operations Plan,* Laboratory for Atmospheric and Space Physics, University of Colorado, document 20517-00-3005.

[4] Carlton, *Real-time Programming,*. New York: Springer, 1988.

Dr. Christopher A. Grasso is a flight software consultant working with the Deep Space Mission Systems directorate of the Jet Propulsion Laboratory. He has developed telemetry, i/o, and sequencing flight components for six JPL deep space missions. He earned a PhD in Electrical and Computer Engineering from the University of Colorado in Boulder for work on provably correct real-time system. He now holds an adjunct faculty position at the University of Colorado to teach spacecraft software systems to undergraduate seniors.

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